UNIVERSIDADE NOVA DE LISBOA
Faculdade de Ciências e Tecnologia
Departamento de Informática

CpPNeTS: uma Classe de Redes de Petri de Alto-nível

Implementação de um sistema de suporte à sua aplicação e análise

Por:
João Paulo Mestre Pinheiro Ramos e Barros

Dissertação apresentada na Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa para obtenção do grau de Mestre em Engenharia Informática

Lisboa
1996

Esta versão electrónica em pdf contém erros de paginação resultantes de dificuldades de conversão do original. Como consequência, os números de página nos índices não estão correctos. Por este facto,aqui ficam as minhas desculpas.

João Paulo Barros

para minha mãe
e em memória de meu pai

Sumário

Nesta dissertação é apresentado um sistema de suporte à aplicação de uma nova classe de Redes de Petri (RdP) de alto-nível (*CpPNeTS*), que permite a geração automática de código para controladores, nomeadamente controladores lógicos programáveis industriais. Inicia-se com uma breve apresentação das RdP, que dá especial ênfase às RdP de alto-nível e às RdP não-autónomas. Seguidamente, apresentam-se as CpPNeTS. Estas são RdP de alto-nível, hierárquicas e temporizadas, com capacidade de especificação de actuações externas e de sincronismo com eventos externos, em função da marcação da rede. A modelação do tempo permite a especificação, análise e simulação de sistemas com exigências de tempo real. A linguagem C++ é utilizada para as inscrições da rede. Define-se uma linguagem de descrição para CpPNeTS (*CpPNeTS-DL*) e descreve-se um pré-processador que permite traduzir a linguagem de descrição em código C++. Esse código é compilado e ligado com uma biblioteca C++ que implementa as funcionalidades do sistema. O programa executável resultante pode gerar uma máquina de estados ou um grafo de ocorrências a partir da rede descrita em CpPNeTS-DL. A biblioteca é facilmente extensível com vista a suportar outros tipos de análise e simulação da rede. Após uma descrição da estrutura e funcionalidades da biblioteca implementada, apresenta-se a aplicação dos programas desenvolvidos a vários exemplos usualmente referidos na literatura.

Abstract

In this master thesis a system based upon a new class of Petri Nets (PN) (named *CpPNeTS*) that allows controllers code generation (namely for industrial Programmable Logic Controllers) is presented. It starts with a brief PN overview with special emphasis on high-level PN and non-autonomous PN. After, CpPNeTS are presented. They are hierarchical and timed high-level PN, with external actuation and synchronisation capabilities, according to net marking. Time modelling allows real-time systems specification, analysis and simulation. The C++ language is used for the net inscriptions. A description language for the CpPNeTS (*CpPNeTS-DL*) is defined and a pre-processor that allows the language translation into C++ code is described. That code is compiled and linked with a C++ library that implements the system functionalities. The resulting executable program can generate a state machine or an occurrence graph after the net specified in the CpPNeTS-DL. The library is easily extensible as to permit further analysis and net simulation. After a explanation of the library structure and functionality the developed code is applied to several examples commonly found in the literature.

Simbologia e Notações

A Matemática pode ser definida como o tema no qual nunca sabemos sobre o que estamos a falar, nem se o que estamos a dizer é verdade.

Bertrand Russel

Geral

$=$	Igual.
$\neq$	Diferente.
$\wedge$	Conjunção lógica (e lógico).
$\{e_1, e_2, e_3, ..., e_n\}$	Conjunto formado pelos elementos e_1, e_2, e_3, ..., e_n. Representação em extensão.
$\{x \mid P(x)\}$	Conjunto de todos os elementos x, tais que P(x) é verdade. Representação em compreensão.
$(a_1, a_2, ..., a_n)$	Sequência de comprimento *n*, ou *n*-tuplo. Duas sequências são iguais se contêm os mesmos elementos na mesma ordem. Assim sendo, um par ordenado é um 2-tuplo. Pressupõe-se a existência de uma relação de ordem *menor que*: a_1 *menor que* a_2 *menor que ... menor que* a_n; ou $a_1 < a_2 < ... < a_n$.
N	Conjunto dos números naturais $\{1, 2, 3, ...\}$.
N_0	Conjunto dos números naturais incluindo o zero: $\{0, 1, 2, 3, ...\}$.
$x \in A$	Relação de pertença. O objecto x é um elemento do conjunto (ou sequência) A.
$x \notin A$	Negação de $x \in A$.
$\sum_{x \in A} m(x)' x$	Multiconjunto de elementos do conjunto A. m(x) denota a quantidade de ocorrências do elemento x de A, no multiconjunto. Assim: $A = \sum_{x \in A} 1' x$. Por exemplo, com $A = \{a, b, c\}$ e $M = \sum_{x \in A} m(x)' x$ podemos ter M = 2'a + 3'b + 1'c.
$\varnothing$	Conjunto vazio.

$\cap$	Intersecção de conjuntos.
$\cup$	Reunião de conjuntos.
A-B	Diferença de conjuntos, isto é, o conjunto formado pelos elementos de A que não pertencem a B: $A-B = \{x: x \in A, x \notin B\}$. Ao conjunto A-B também chamaremos: complementar de B em A.
A-x	É o conjunto de todos os elementos de A excepto x: $A-x = \{y \mid y \in A \wedge y \neq x\}$.
$A \times B$	Produto cartesiano: $\{ (x, y) \mid x \in A, y \in B \}$.
$A \subseteq B$	A está contido em B. Todo o elemento de A é também elemento de B.
2^A	Conjunto de todos os subconjuntos de A: $2^A = \{X \mid X \subseteq A\}$.
$g \subseteq A \times B$	Relação de A para B.
$f: A \rightarrow B$	Aplicação f de A em B. Função f definida em A e com valores em B.
f(x)	Valor de f em x. Para $f: A \rightarrow B$, $x \in A$ e $f(x) \in B$.
Var(E)	Função que retorna o conjunto das variáveis presentes em E.
$<v_1 \leftarrow d_1, ..., v_n \leftarrow d_n>$	Substituição ou vínculo. Correspondência de cada variável v_i a um valor d_i. Diz-se também que a variável v_i se encontra vinculada a d_i ou que é substituída por d_i.

Redes de Petri

L	Conjunto dos lugares de uma RdP
T	Conjunto das transições de uma RdP
A	Conjunto dos arcos de uma RdP
•l	Com $l \in L$, é o conjunto das transições de entrada de l.
l•	Com $l \in L$, é o conjunto das transições de saída de l.
•t	Com $t \in T$, é o conjunto dos lugares de entrada de t.
t•	Com $t \in T$, é o conjunto dos lugares de saída de t.

Nota sobre a simbologia utilizada:

Utilizaram-se letras maiúsculas para denominar conjuntos. Para evitar a utilização de nomes pouco significativos, optou-se por utilizar também nomes de conjuntos constituído por duas ou mais letras maiúsculas podendo alguns apresentar um índice subscrito, por exemplo: C, AE, SD_C são todos possíveis nomes de conjuntos.

Índice de Matérias

Índice de Figuras

Índice de Quadros

Introdução

A maior parte dos sistemas dinâmicos que encontramos na Natureza podem ser descritos por equações diferenciais em que o tempo conduz a evolução do sistema. Esses sistemas são estudados, desde os tempos de Galileu e Newton existindo uma extensa bibliografia a par com numerosas aplicações de sucesso [Ho, 92b: vii]. No entanto, a tecnologia moderna tem vindo a criar sistemas dinâmicos que não são facilmente descritos por equações diferenciais. Estes sistemas apresentam uma diferença fundamental relativamente aos que podemos encontrar na Natureza: são conduzidos por eventos assíncronos e não pelo fluxo continuo do tempo, ou seja, apenas modificam o seu estado aquando da ocorrência de determinados eventos. Como exemplos de sistemas deste tipo temos: uma linha de montagem de uma fábrica automatizada, um elevador de um prédio ou uma rede de comunicações que envia e processa pacotes de informação. Todos eles têm em comum a existência de um conjunto de recursos (máquinas, elevadores, linhas de comunicação) e de tarefas (fabricar automóveis, enviar elevador para o andar x, enviar pacotes de informação). As tarefas deslocam-se de recurso para recurso competindo pelos serviços. Estes sistemas são normalmente denominados: sistemas dinâmicos a eventos discretos ou, simplesmente, sistemas a eventos discretos (SED) [Ho, 92a].

Comparando com o paradigma das equações diferenciais para os sistemas dinâmicos de variáveis contínuas, a modelação matemática dos SED encontra-se ainda na sua infância [Ho, 92b: vii]. Existem já muitas tentativas de obtenção de modelos gerais para este tipo de sistemas. No entanto,

não existe ainda um consenso sobre qual o modelo que apresenta o potencial de servir de analogia à utilização de equações diferenciais nos sistemas dinâmicos de variáveis continuas. Em [Ho, 92b: vii] apresenta-se um resumo dos vários tipos de modelação de SED.

Um sistema de manufactura flexível é um exemplo típico de um sistema a eventos discretos. Como a análise e controlo desses sistemas é de grande importância prática, grande parte dos formalismos de controlo de SED são apresentadas como resolvendo (ou contribuindo para) esses objectivos. No entanto, existe ainda uma grande distância entre as promessas teóricas e as aplicações práticas. Por um lado muitos dos formalismos teóricos são aplicados a exemplos triviais, por outro lado, os sistemas a eventos discretos reais são implementados com poucas contribuições da teoria [Inan et al., 88: 627]. Para o controlo deste tipo de sistemas, utilizam-se, normalmente, controladores lógicos programáveis (PLCs[1]). O PLC é um dispositivo, relativamente pouco dispendioso, para controlo flexível que pode ser considerado como o dispositivo normalizado de controlo para processos individuais, bem como, para a integração de processos. Um PLC pode ser visto como um computador dedicado ao controlo de processos. Os controladores lógicos programáveis são frequentemente utilizados para o controlo sequencial de sistemas automatizados, nos quais se incluem os sistemas flexíveis de manufactura. Apesar de actualmente existirem já PLCs que suportam linguagens de programação de alto nível, a programação utilizando diagramas de contacto continua a ser a mais utilizada [Chang et al., 91][2]. Os diagramas de contacto constituem uma linguagem de muito baixo nível de abstracção sofrendo das desvantagens inerentes a esse tipo de linguagens: os programas são difíceis de corrigir e modificar tornando a seu desenvolvimento e manutenção menos eficientes do que o desejado. De facto, estes diagramas atingem dimensões tais que tornam extremamente difícil a detecção de erros. Para além desse facto, a sua utilização encontra-se limitada ao controlo do sistema, não permitindo a análise e/ou simulação do sistema [Venkatesh et al., 95]. Com a notável excepção, no mercado francês, do Grafcet [David et al., 92][Novais, 94], as alternativas aos diagramas de contacto são constituídas por dialectos de uma das linguagem de programação imperativa mais comuns como C, PASCAL, BASIC ou Assembly [Gomes, 93]. Estas apresentam a desvantagem de constituírem "soluções de construtor", ou seja,

[1] Do Inglês ***P**rogrammable **L**ogic **C**ontroller*.

[2] Esta referência oferece uma breve introdução à programação de PLCs, utilizando diagramas de contacto.

tipicamente cada construtor de PLCs oferece uma linguagem para a programação específica dos seus controladores. Quanto ao Grafcet, está formalmente definido numa norma francesa denominada "Grafcet - Diagrama funcional para a descrição de sistemas lógicos de comando". No entanto, a sua adopção em ambientes industriais não pode ainda ser considerada muito significativa [Gomes, 93].

Na prática, além dos diagramas de contacto, apenas as listas de instruções (linguagens tipo Assembly) são largamente utilizadas. Estas são, também, linguagens de baixo-nível pelo que apresentam desvantagens semelhantes às referidas paras os diagramas de contacto. Resta referir que existe um esforço de normalização em curso para as linguagens de programação de PLCs [IEC, 92].

Entre os vários tipos de modelação de SED e respectivo controlo, encontram-se os modelos baseados em autómatos entre os quais se contam as máquinas de estado e seus derivados, como por exemplo, os diagramas de transição de estado, *Statecharts* [Harel, 87] e redes de Petri (daqui em diante referenciada apenas RdP) [Murata, 89]. As RdP, apresentadas no capítulo 1, são um tipo particular de grafos bipartidos dirigidos que oferecem um ambiente uniforme para a modelação, análise formal e simulação de SED. Actualmente existem já muitas variantes do modelo seminal de RdP[3].

A aplicação de RdP à realização de controladores industriais tem já um longo historial. Em [Zurawski et al., 94: 568] e [Zhou, 95: 4], encontram-se listas de referências sobre trabalhos de síntese e implementação de controladores sequenciais baseados em RdP. Estas permitem outra forma de especificação de programas para PLCs, sem as dificuldades características da utilização dos diagramas de contacto. Em [Murata, 95] são apresentados vários modelos convencionais para a especificação de controlo sequencial, sendo, no final, apresentado um modelo baseado numa nova classe de RdP: as RdP Evento/Predicado. Em [Venkatesh et al., 95] é feita uma comparação entre o desenho do controlo de sistemas a eventos discretos utilizando diagramas de contacto e RdP[4].

[3] No apêndice A encontra-se uma listagem das mais frequentemente encontradas.

[4] O método de controlo do sistema utilizando uma RdP recorre à execução da mesma, que controla o sistema em tempo-real. A classe de RdP apresentada é semelhante à das RdP ordinárias, e é denominado *Real Time Petri Nets*.

Também neste trabalho pode encontrar-se uma tabela dos vários esquemas de implementação de controladores sequenciais baseados em RdP e respectivas referências na literatura.

Em [Zhou, 92], é apresentada uma arquitectura para um gerador de programas de controlo automático. Esta apresenta objectivos idênticos ao sistema aqui proposto mas recorre a RdP ordinárias: a RdP é especificada num editor gráfico e pode ser traduzida para código de controlo servindo-se da informação contida numa base de dados e numa base de conhecimentos.

Outra estratégia para o controlo do sistema a eventos discretos, consiste na interpretação da rede. Nesse caso, as acções no sistema são desencadeadas pela decisão de quais as transições a disparar em cada instante. Este mecanismo, obriga a uma resposta em tempo-real por parte do executor da RdP, normalmente um interpretador. O tempo de resposta vai depender do sistema, e no caso de redes complexas e/ou sistemas que exijam tempos de resposta muito curtos, esta opção torna-se, rapidamente, inviável. Embora à partida, a utilização de RdP de baixo-nível permita uma grande eficiência da parte do interpretador [Silva, 85], também é verdade que tal obrigará à criação de uma rede de grande dimensão para modelar qualquer sistema não trivial.

Em [Shukla et al., 91] e [Wongtaladkown et al., 90] são descritos programas para a edição e simulação gráficas de RdP ordinárias. Em [Wongtaladkown et al., 90] existe uma maior preocupação com sistemas tempo-real, mas o sistema é muito limitado, suportando um máximo de setenta nós na rede.

Apesar de actualmente serem reconhecidas significativas vantagens das RdPCol face às RdP de baixo-nível, todas estas aproximações ao controlo de PLCs se baseiam em RdP baixo-nível[5]. Tipicamente estas soluções recorrem a técnicas de implementação directa de RdP em controladores de baixo custo [Silva, 85] ou computadores [Silva, 85][Zhou, 92]. Tais implementações são relativamente simples de obter mas o mesmo não se pode dizer da especificação a efectuar pelo desenhador. Com efeito, as limitações das RdP de baixo-nível, reflectidas na grande quantidade de nós da rede necessários, vêm rapidamente ao de cima quando se pretende especificar um sistema não trivial. As RdP de alto-nível permitem uma muito maior facilidade e simplicidade de

[5] No capítulo 1 apresentam-se detalhadamente umas e outras. Resumidamente podemos afirmar que as RdP de alto-nível estão para as RdP de baixo-nível assim como as linguagens de programação de alto-nível

especificação mas a sua execução é, normalmente, demasiado exigente para o controlo "directo" de sistemas em tempo-real.

Em [Gomes et al., 92] foi apresentada uma metodologia (actualizada em [Gomes et al., 95]) para especificação e implementação de controladores lógicos programáveis para sistemas em que condicionalismos de tempo real e concorrência de acções são características principais. Essa metodologia baseia-se na utilização de RdP coloridas e sincronizadas às quais se adicionaram temporizações, associadas às marcas. Aí é realçado o facto de a programação de PLCs exigir um conhecimento detalhado de cada PLC que se pretenda programar, bem como, do ambiente de simulação/emulação associado para validação do sistema. Como resposta a esse problema, é apresentado um ambiente de desenvolvimento integrando a metodologia descrita.

O programa editor-simulador descrito em [Gomes et al., 94] é o único, do conhecimento do autor, a utilizar RdP de alto-nível e sincronizadas com vista à modelação de controladores sequenciais. Esse programa insere-se no ambiente de desenvolvimento, apresentado em [Gomes et al., 93], para suporte à metodologia descrita. Essa metodologia apresenta dois objectivos principais: utilização de um formalismo de especificação e análise do funcionamento do controlador uniforme para todos os tipos de controlador e o deslocamento para o sistema de desenvolvimento do maior número de tarefas com vista a possibilitar a utilização de controladores de baixo custo. O sistema é constituído, fundamentalmente, por um editor gráfico, um simulador, um analisador e um *assemblador*[6]. O editor permite a construção interactiva de uma classe de RdP coloridas e sincronizadas que modela o controlador. O simulador permite o teste *offline* da especificação realizada, e o analisador é o responsável pela tradução da rede de Petri utilizada na especificação, numa máquina de estados. Esta máquina de estados pode então ser traduzida para a linguagem específica de cada PLC, pelo *assemblador*. Esta aplicação é a única que necessita conhecer as especificidades do PLC onde será implementada a máquina de estados..

---

estão para a linguagem máquina (ou linguagem *assembly* se incluirmos a notação associada a cada elemento da rede).

[6] Optou-se por manter a designação utilizada em [Gomes et al., 93].

O sistema desenvolvido no âmbito desta dissertação, denominado CpPNeTS-S (*C++ based Coloured Petri Nets with Time and Synchronisation System*)[7], baseia-se, fundamentalmente, no proposto em [Gomes et al., 93], mais especificamente na aplicação *analisador*. Partindo de uma especificação da rede, obtem-se uma máquina de estados, realizável num controlador específico, conhecidas as suas características específicas. O sistema é, no entanto, muito mais genérico do que o analisador apresentado em [Gomes et al. 93]. De facto, a geração da máquina de estados é apenas um dos procedimentos realizável sobre o suporte oferecido. A especificação da rede é traduzida para uma representação sobre a qual é possível operar de forma a efectuar outras operações, designadamente técnicas de análise e simulação, comuns às RdPCol[8]. Algumas das técnicas de análise realizáveis tais como: construção de grafos de ocorrências [Jensen, 95: 1-98] e análise de invariantes [Jensen, 90: 147-50][Jensen, 95: 101-38], encontram-se já bem estudadas para as RdPCol, e são directamente aplicáveis às CpPNeTS. O mesmo sucede com outras técnicas menos estudadas, tais como, regras de redução [Jensen, 90: 150-1] e análise de desempenho [Jensen, 90: 152]. As técnicas de simulação podem também ser implementadas com base na representação da rede utilizada. Nomeadamente, a análise de desempenho pode ser feita não só de um modo formal (por exemplo, através da utilização de cadeias de Markov [Jensen, 90: 152]), mas também recorrendo a simulação, o que para redes mais complexas se pode revelar de grande importância.

Uma nova classe de redes de Petri de alto nível (as CpPNeTS) baseada em trabalhos já realizados [Gomes et al., 92][Gomes et al., 93][Gomes et al., 95], permite, designadamente, o suporte e implementação do referido sistema. As CpPNeTS apresentam-se particularmente bem adaptadas à utilização como ferramentas de construção de programas para PLCs, mais concretamente, porque:

- São redes de alto-nível (coloridas) hierárquicas, permitindo uma especificação muito mais compacta e legível quando comparadas com as RdP de baixo-nível utilizadas em sistemas semelhantes referidos na literatura [Zhou, 92].

[7] Optou-se por um acrónimo na língua inglesa como forma de facilitar a divulgação do sistema.

[8] Se se pretender recorrer às técnicas já estudadas para as RdPCol [Jensen, 95], bastará omitir as extensões permitidas pelas CpPNeTS.

- Oferecem a possibilidade de especificação da sincronização com eventos externos (nos disparos das transições).
- Oferecem a possibilidade de especificação de actuações externas (invocação de procedimentos) em função da evolução da marcação da RdP (disparo das transições) e das marcas presentes nos lugares.
- São temporizadas. Esta característica permite a modelação, análise e simulação de sistemas com exigências de tempo real.

A junção das características de uma RdP de alto-nível, com as de uma RdP temporizada torna possível a representação de relações entre os aspectos funcionais e temporais do modelo, mais especificamente é possível fazer com que as temporizações dependam da marcação da rede.

Dado que a especificação de característica não-autónomas da RdP (sincronização e temporização), bem como, das actuações externas, é sempre opcional, as CpPNeTS, são suficientemente gerais para abarcarem o tipo de modelação permitido pelas RdP coloridas hierárquicas e temporizadas (§1.7.1 §1.7.2 §1.7.3). Desta forma, a implementação efectuada abre caminho a futuras realizações utilizando a mesma representação da rede.

Estrutura da tese

No capítulo 1 apresentam-se as redes de Petri. Actualmente estão facilmente disponíveis várias excelentes introduções ao estudo das RdP [Peterson, 77][Peterson, 81][Silva, 85][Murata, 89][David, 91][Reisig, 92][Zurawsky et al., 94][Zhou, 95], razão pela qual se optou por uma apresentação centrada maioritariamente nos aspectos fundamentais (estrutura e comportamento), em detrimento de outros, menos significativos para o presente trabalho, designadamente, as questões de análise da rede e suas propriedades estáticas e dinâmicas. Estas questões, que podem revelar-se muito importantes em futuras ampliações do sistema desenvolvido, e que são certamente fundamentais para uma mais completa compreensão do funcionamento e aplicabilidade das RdP, encontram-se extensamente tratadas na bibliografia citada. Seguidamente,passam-se em revista os três modelos de RdP de alto-nível, mais citados na literatura. As RdP de alto-nível, nas quais se incluem as CpPNeTS, apresentam a vantagem de possuírem uma maior capacidade de compactação do modelo e consequente legibilidade, à custa de uma maior complexidade de análise e implementação. Na parte final do capítulo, apresenta-se também a possibilidade de especificação

hierárquica das RdP e evidencia-se o facto de tal ser mais vantajoso quando aplicado às RdP de alto-nível. O modelo seminal das RdP não contempla a modelação de tempo. Actualmente existem vários modelos de RdP com temporizações. Os mais significativos são sucintamente apresentados. O capítulo termina com uma breve referência a uma outra classe de RdP, as RdP não-autónomas que incluem as RdP temporizadas e também as RdP sincronizadas. Nestas o disparo das transições encontra-se condicionado pela ocorrência de eventos externos ao modelo.

No capitulo 2 define-se formalmente uma nova classe de RdP de alto-nível (as CpPNeTS), bem como o seu comportamento. É dada especial enfase às características que tornam esta classe de redes especialmente adequada à modelação do controlo de sistemas a eventos discretos, designadamente as capacidades de especificação de eventos e actuações externos. É apresentada, de forma simplificada, a relação entre a estrutura de um controlador programável e a sua representação utilizando uma CpPNeTS. Compara-se o sistema implementado (CpPNeTS-S) para a representação, análise e simulação de redes desse tipo, com outros semelhantes referidos na literatura. O sistema recorre a um programa que traduz a especificação de cada página da CpPNeTS em código C++. Este código é posteriormente compilado e ligado a uma biblioteca pré-compilada de classes C++ que fornecem o suporte para a representação da rede, bem como para os vários algoritmos de análise e simulação. Por fim, apresentam-se as razões da escolha da linguagem C++ não só para a implementação do sistema mas também para as anotações das redes.

A especificação das CpPNeTS é actualmente realizada recorrendo a uma linguagem descritiva. No capítulo 3 descreve-se um programa que possibilita a tradução dessa especificação em classes C++. Trata-se de um pré-processador cujo código produzido é posteriormente processado por um compilador da linguagem C++. Posteriormente apresentam-se as várias construções da linguagem, paralelamente com o respectivo código gerado pelo tradutor. O apêndice D contem a gramática da linguagem.

No capitulo 4 surge a descrição da biblioteca de classes C++ implementada para suporte ao sistema. A biblioteca é constituída por um conjunto de classes C++ pré-compiladas. Estas destinam-se a ser ligadas ao código resultante da compilação das classes geradas pelo tradutor. As classes da biblioteca fornecem o suporte à representação da rede, bem como um conjunto de procedimentos de simulação e análise da mesma. A rede é definida pelas classes traduzidas pelo pré-processador. A versão actual da biblioteca oferece dois procedimentos de análise da rede: o

gerador da máquina de estados síncrona de uma CpPNeTS e o gerador do grafo de ocorrências de uma CpPNeTS autónoma. Descreve-se o suporte fornecido pela biblioteca, para a representação hierárquica da rede, bem como dos constituintes de cada página: lugares, transições, arcos e contendas §2.1.1 (uma nova estrutura que agrupa transições constituintes de um mesmo conflito estrutural). Seguidamente, descreve-se a determinação dos vínculos da rede e respectivas representações, bem como, a forma como essas representações são utilizadas aquando do disparo da rede. As estruturas necessárias para suporte à temporização e sincronização da rede são também apresentadas. O capítulo termina com a descrição dos algoritmos de análise já implementados: construção do grafo de ocorrências de uma rede autónoma e a construção da máquina de estados síncrona.

No capítulo 5 são apresentados vários exemplos de modelação utilizando o sistema CpPNeTS-S. Cada um dos exemplos permite ilustrar alguma capacidade que não surge nos restantes. Para cada exemplo apresenta-se a especificação da rede utilizando a linguagem CpPNeTS-DL e os respectivos resultados, obtidos após execução do programa executável. O código C++, resultante da aplicação do pré-processador a algumas das especificações, encontra-se no apêndice E, como forma de minorar a extensão do capítulo.

No capítulo 6 tecem-se alguns comentários sobre os resultado obtidos e sugerem-se possíveis adições e melhoramentos ao sistema desenvolvido.

Capítulo 1

Redes de Petri

E agora para algo completamente diferente
Monty Python's Flying Circus

As redes de Petri (RdP) devem o seu nome ao trabalho de Carl Adam Petri que na sua dissertação de doutoramento, submetida, em 1962, à Faculdade de Matemática e Física da Universidade Técnica de Darmstadt na Alemanha, apresentou um tipo de grafo bipartido[9] com estados associados, com o objectivo de estudar a comunicação entre autómatos [Murata, 89][10]. O seu desenvolvimento posterior foi catalisado pelas suas numerosas potencialidades de modelação, designadamente: sincronização de processos, concorrência, conflitos e partilha de recursos. Actualmente, decorreram já mais de trinta e três anos de trabalhos teóricos e aplicações sobre RdP tendo este estudo levado, quer a um desenvolvimento das técnicas de análise das RdP e sua aplicação prática, quer ao desenvolvimento de muitas variantes ao modelo seminal das RdP tendo em vista aplicações específicas.

Como ferramentas matemáticas e gráficas, as RdP oferecem um ambiente uniforme para a modelação, análise formal e simulação de sistemas a eventos discretos, permitindo uma visualização simultânea da sua estrutura e comportamento.

Mais especificamente, as RdP modelam dois aspectos desses sistemas, eventos e condições, bem como, as relações entre eles. Segundo esta caracterização, em cada estado do sistema verificam-se determinadas condições. Estas podem possibilitar a ocorrência de eventos que por sua vez podem

[9] Um grafo G denomina-se bipartido quando os seus nós podem ser divididos em dois conjuntos N_1 e N_2, tais que nenhum nó contido em N_1 ou N_2 se encontra ligado a outro nó contido no mesmo conjunto.

ocasionar a mudança de estado do sistema [Peterson, 77: 228]. Como veremos, é possível relacionar, de uma forma intuitiva, condições e eventos com os dois tipos de nós da rede, respectivamente lugares e transições.

1.1 Constituição e estrutura de uma Rede de Petri - lugares, transições, arcos e marcas

Os elementos dos dois conjuntos em que se podem dividir os nós constituintes de uma RdP denominam-se, respectivamente, *lugares* e *transições*. Os lugares são normalmente representados por circunferências ou elipses, e as transições por segmentos de recta, rectângulos ou barras. Os lugares encontram-se ligados às transições, e estas aos lugares, através de arcos dirigidos (Figura 1.1).

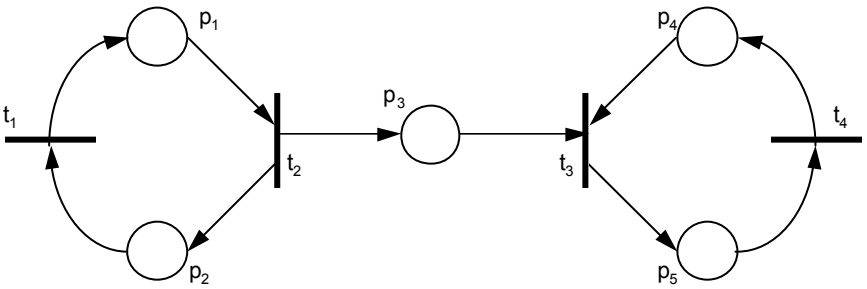


Figura 1.1 Uma rede de Petri.

A estrutura de uma rede de Petri pode, pois, ser definida como um tuplo R = (L, T, AE, AS) [Peterson, 77: 235]conforme apresentado no Quadro 1.1.

[10] Este trabalho contêm um breve resumo histórico sobre a origem das RdP.

R = (L, T, AE, AS) onde:
$L = \{ p_1, p_2, ..., p_m \}$ é um conjunto de lugares [11]
$T = \{ t_1, t_2, ..., t_n \}$ é um conjunto de transições
$L \cap T = \varnothing$ os conjuntos L e T são disjuntos
AE: $L \times T$ é um conjunto de arcos de entrada nas transições
AS: $T \times L$ é um conjunto de arcos de saída das transições

Por exemplo, para a Figura 1.1 temos:
$L = \{ p_1, p_2, p_3, p_4, p_5 \}$
$T = \{ t_1, t_2, t_3, t_4 \}$
$AE = \{ (p_2, t_1), (p_1, t_2), (p_3, t_3), (p_4, t_3), (p_5, t_4) \}$
$AS = \{ (t_1, p_1), (t_2, p_2), (t_2, p_3), (t_3, p_5), (t_4, p_4) \}$

Quadro 1.1 Definição de uma Rede de Petri.

Dado uma RdP constituir um modelo abstracto, podem ser dadas várias interpretações distintas a lugares e transições, conforme o tipo de aplicação em causa. No entanto, dadas as suas características intrínsecas, os lugares podem, de uma forma muito genérica e muito próxima da sua visualização gráfica, ser vistos como depósitos de recursos e as transições como acções que manipulam esses recursos. Os recursos são representados gráficamente por pequenos circulos pretos dentro dos lugares. A cada um desses círculos dá-se o nome de *marca*. Os lugares surgem como representantes do estado da rede e as transições como responsáveis pela mudança de estado; desta forma, a RdP é simultaneamente orientada para os estados e para as acções. Ao estado da rede é usual chamar-se marcação, dado ser definido pela marcação de cada um dos seus lugares, ou seja, pela quantidade de marcas presentes em cada lugar. As marcas são representadas graficamente como pequenos círculos pretos, dentro dos lugares. A função das transições consiste em destruir e/ou criar estas marcas. Como as transições estão obrigatoriamente entre lugares é através da sua acção (denominada disparo) que um lugar altera a sua marcação. Os arcos indicam, para cada transição, os lugares sobre os quais estas actuam. Na Figura 1.2 vemos um exemplo de uma RdP marcada[12] correspondente à RdP da Figura 1.1

[11] Dado a a letra "l" ser facilmente confundida com o algarismo um ("1") optou-se pela utilização da letra "p" para denominar elementos do conjunto "L".

[12] Por enquanto, podemos ignorar ou números que surgem junto de três dos arcos.

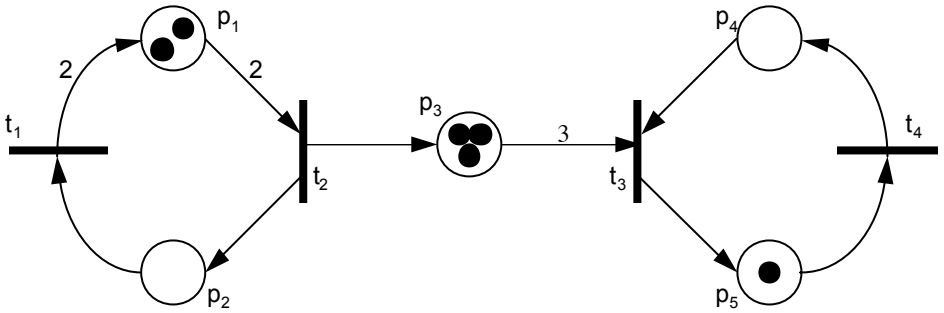


Figura 1.2 Uma rede de Petri marcada.

Sendo as marcações nos lugares as representantes dos estado da rede, as transições surgem como os agentes que fazem a rede evoluir de estado para estado. Seguidamente ver-se-à como se processa essa evolução.

1.2 A evolução da rede — o disparo das transições

Conforme já referido, são as transições que podem destruir e criar as marcas contidas nos lugares. Os arcos ligados a cada transição indicam exactamente sobre que lugares actuam. Um arco com origem num lugar e término numa transição (a partir daqui designado por arco de entrada), indica que essa transição subtrai, aquando do seu disparo, uma marca desse lugar. De forma simétrica, um arco com origem numa transição e fim num lugar (daqui em diante designado por arco de saída), indica que essa transição adiciona, aquando do seu disparo, uma marca a esse lugar. Assim sendo, podemos pensar nos arcos como indicando o sentido do movimento das marcas de um lugar para outro, atravessando a transição.

Daí resulta que uma transição só pode disparar se cada lugar de entrada contiver pelo menos uma marca, de forma a poder ser retirada no disparo da transição, pelo respectivo arco. Quando tal sucede diz-se que a transição está *habilitada*[13]. O disparo de uma transição é instantâneo, ou seja, as duas acções citadas são efectuadas ao mesmo tempo. Assim sendo, não existe nenhum instante no qual todos, ou parte, dos lugares de entrada já contêm menos uma marca e todos, ou parte, dos lugares de saída ainda não contêm mais uma marca, vide Figura 1.3.

[13] *enabled* na bibliografia em língua inglesa.

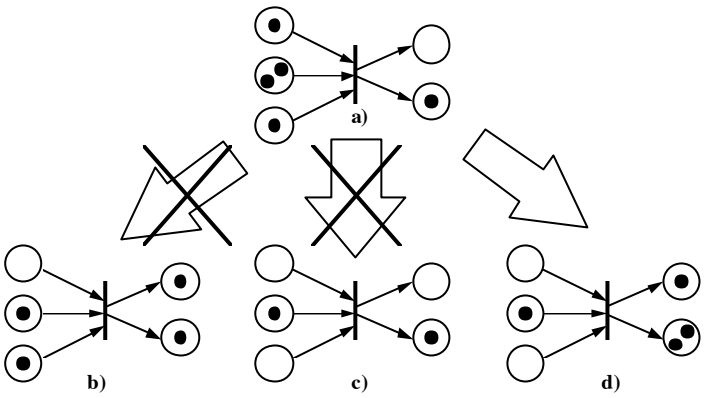


Figura 1.3 Disparo de uma transição. a) corresponde à situação inicial. b) e c) não correspondem ao resultado do disparo da transição. Apenas d) representa a rede obtida após o disparo.

É muito importante notar que apesar de, por vezes, se falar em "movimento de marcas", tal é incorrecto do ponto de vista formal. O que acontece, aquando do disparo de uma transição, não é a mudança de marcas, dos lugares de entrada da transição para os lugares de saída, mas sim, a remoção de marcas nos lugares de entrada e a criação de novas marcas nos lugares de saída. Os arcos de saída são, portanto, criadores de novas marcas e não simples depositantes das marcas retiradas pelos arcos de entrada. Tal seria contraditório com o facto de uma transição poder possuir diferentes quantidades de arcos de entrada e de saída.

Um tipo de redes[14], muito frequente, denominado RdP generalizada, permite a existência de múltiplos arcos entre nós. Tal equivale a associar um inteiro a cada arco, constituindo, dessa forma, uma generalização do tipo de arcos já descrito. Nessas redes, cada arco não "transporta" necessariamente uma marca mas sim a quantidade especificada. A essa quantidade associada a cada arco dá-se o nome de *peso*.

A RdP até aqui descrita corresponde à apresentada em [Peterson, 77: 235]. Conforme já aí referido: "A maioria dos investigadores utilizam Redes de Petri generalizadas nos seus trabalhos, muitas vezes ignorando a distinção entre elas e aquelas que definimos como Rede de Petri" [Peterson, 77: 246]. Por exemplo, em [Murata, 89: 543] o autor apresenta, com a designação de Redes de Petri uma Rede de Petri generalizada. Por outro lado, a marcação surge como parte da definição de RdP, não sendo feita a distinção entre RdP e RdP marcada. Daí resulta uma definição

[14] O glossário do Apêndice A apresenta alguns dos principais tipos de RdP em ordem alfabética, bem como uma sua breve descrição informal.

mais extensa (Quadro 1.2) onde, para além dos pesos dos arcos, se inclui já a marcação. Por outro lado, efectuou-se uma compactação ao reunir os conjuntos relativos aos arcos.

$R = (L, T, A, PA, M_0)$ onde:

$L = \{ p_1, p_2, ..., p_m \}$ é um conjunto de lugares
$T = \{ t_1, t_2, ..., t_n \}$ é um conjunto de transições
$L \cap T = \varnothing \quad \wedge \quad L \cup T \neq \varnothing$ os conjuntos L e T são disjuntos e não vazios
$A: (L \times T) \cup (T \times L)$ é o conjunto dos arcos
$PA: A \rightarrow N$ são os pesos dos arcos
$M_0: L \rightarrow N_0$ é a marcação inicial

Por exemplo, para a Figura 1.2 temos:
$L = \{ p_1, p_2, p_3, p_4, p_5 \}$
$T = \{ t_1, t_2, t_3, t_4 \}$
$A = \{ (p_2, t_1), (p_1, t_2), (p_3, t_3), (p_4, t_3), (p_5, t_4), (t_1, p_1), (t_2, p_2), (t_2, p_3), (t_3, p_5), (t_4, p_4) \}$

$PA((p_2, t_1)) = 1$	$PA((t_1, p_1)) = 2$	$M_0((p_1) = 2$
$PA((p_1, t_2)) = 2$	$PA((t_2, p_2)) = 1$	$M_0((p_2) = 0$
$PA((p_3, t_3)) = 3$	$PA((t_2, p_3)) = 1$	$M_0((p_3) = 3$
$PA((p_4, t_3)) = 1$	$PA((t_3, p_5)) = 1$	$M_0((p_4) = 0$
$PA((p_5, t_4)) = 1$	$PA((t_4, p_4)) = 1$	$M_0((p_5) = 1$

Quadro 1.2 Definição de uma Rede de Petri generalizada (com pesos) com marcação inicial.

Com base nesta definiçao diz-se que uma transição se encontra habilitada se e só se:

$$(\forall (p, t) \in A)\ PA((p, t)) \leq M(p).$$

Daqui em diante uma Rede de Petri generalizada e marcada será denominada simplesmente por Rede de Petri ou RdP.

Tal como o sistema modelado, a RdP não prevê nem impõe uma ordem de ocorrência dos eventos. Tal significa que se em determinado instante, mais do que uma transição se encontra apta a disparar, qualquer uma pode disparar. O estado seguinte é resultado do disparo de um qualquer multiconjunto[15] de transições habilitadas. A este multiconjunto de transições aptas dá-se o nome de passo. A quantidade de estados seguintes possíveis corresponde pois à quantidade existente de diferentes passos.

[15] Um multiconjunto é uma estrutura idêntica a um conjunto mas com uma única e importante diferença: pode conter mais do que uma ocorrência de um mesmo elemento.

1.3 Modelação com Redes de Petri

Quando modelamos um sistema através de uma RdP, estamos necessariamente a criar uma interpretação da rede[16]. É essa interpretação ou significação que efectua a ligação do modelo abstracto que qualquer RdP representa, com o sistema concreto que se pretende modelar. Por exemplo, uma possível interpretação da rede da Figura 1.2, é a que se apresenta na Figura 1.4 onde temos a modelação de um sistema produtor-consumidor [Reisig, 92].

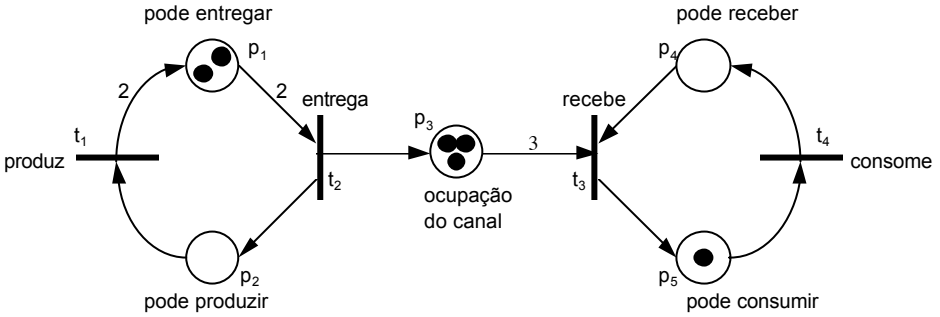


Figura 1.4 Rede de Petri com pesos associados aos arcos (RdP generalizada) e marcada, com interpretação.

Podemos ver que os produtos são produzidos aos pares e que para cada um destes pares é entregue uma unidade. Por outro lado, o consumidor (modelado pelos lugares p_4 e p_5 e pelas transições t_3 e t_4) necessita de receber três unidades e, posteriormente, de as consumir, antes de poder receber mais unidades.

As RdP permitem a modelação e visualização de diversos conceitos e relações, designadamente: paralelismo e concorrência, partilha de recursos, sincronização, memorização, limitação de recursos e leitura. Em [David et al., 92: 64-7] é possível encontrar as construções que permitem esse tipo de modelações.

Além do fluxo de controlo, as RdP podem também representar o fluxo de dados (*dataflow*). Num modelo de fluxo de dados os operadores são activados pela chegada dos operandos. Numa RdP os

[16] Interpretação no sentido de significação. Não confundir com "RdP interpretada". Essa designação pode ser utilizada com vários sentidos [Gomes, 91]. Por exemplo em [Peterson, 77] uma rede não-intepretada é uma rede em que não se atribuiram significados aos lugares, transições, etc.. Nesse caso apenas estamos interessados nas caracteríticas abstractas da rede. A rede não tem significado concreto. Já em [David, 91], uma rede interpretada denomina uma classe específica de redes (ver Apêndice A).

operandos são representados pela presença de marcas nos lugares e os operadores estão associados a transições.

As RdP não oferecem apenas uma representação para a estrutura e funcionamento de um sistema. Permitem a visualização do comportamento do sistema através do "movimento" das marcas. Conforme referido por Robert Valette[17], as RdP capturam a dinâmica dos sistemas a eventos discretos tornando-se dessa forma particularmente úteis para a sua simulação.

1.4 Representação de Redes de Petri

As RdP são grafos. Os grafos são estruturas de dados clássicas cujas técnicas de representação são já bem conhecidas [Sedgewick, 88] [Cormen et al., 90]. Existem duas formas fundamentais de representar grafos: através de uma matriz de incidências ou utilizando uma lista de adjacências. Como nas RdP, é frequente muitos dos nós não se encontrarem ligados entre si, a lista de adjacências constitui, em principio, uma melhor opção tendo em vista a poupança de memória. Consideram-se duas perspectivas quando à melhor representação de uma RdP: "a perspectiva do simulador/executor" e a "perspectiva do editor".

Para o simulador/executor, as transições constituem os elementos da rede que apresentam maior importância. São as transições que determinam a evolução da rede pelo que a estrutura que é necessário percorrer é a que contem as transições. Como para determinar a habilitação, ou não, das transições, são necessárias as marcações dos seus lugares de entrada e os pesos dos seus arcos de entrada, será conveniente que cada transição contenha a lista dos seus arcos de entrada que por sua vez contêm o lugar a que se encontram ligados. Os lugares contêm as respectivas marcações (quantidade de marcas). As transições conterão, também, a lista dos seus lugares de saída que por sua vez contêm o lugar respectivo. Importa assinalar que estas relações de inclusão podem corresponder apenas à inclusão de referências. Por exemplo, cada arco pode conter apenas uma referência para o lugar respectivo que se encontra numa lista de lugares. Essa lista pode ser percorrida de forma a consultar a marcação da rede de forma mais eficiente.

---

[17] Em resposta à pergunta "Que propriedades das RdP as tornam utilizáveis na simulação?" colocada na petri net mailing list por Andreas Mittermayr (mitterma@fmi.uni-passau.de, andreas@newciv.org).

Em [Silva, 85: 315-72] são apresentadas várias representações possíveis (na "perspectiva do simulador/executor") para RdP seguras. Todas procuram maximizar a relação *rapidez/ocupação de memória*.

Na segunda perspectiva considerada, isto é, para um editor gráfico de RdP que interage com os elementos gráficos constituintes, não existe a noção de maior ou menor importância de um tipo de nó da rede face a outro. A adição ou remoção de lugares ou transições é igualmente custosa pois pode ser necessário remover um conjunto de arcos em ambos os casos. Ou seja, para o editor o facto do grafo ser bipartido apenas se reflecte na necessidade de evitar ligações inválidas de arcos. Interessa encontrar, inserir e/ou remover nós da rede. Como à medida que se adiciona mais informação a estrutura se torna mais complexa esse tipo de operações pode dar origem a apontadores que deixam de apontar para endereços válidos (*dangling pointers*). Uma das formas de evitar essa situação consiste em adicionar mais informação à estrutura, nomeadamente transformando listas ligadas simples em listas ligadas duplamente. Outra forma, consiste em marcar os elementos a remover e seguidamente percorrer a estrutura reconstruindo-a[18].

1.5 Propriedades das Redes de Petri

As propriedades das RdP encontram-se extensamente documentadas [Peterson, 77][Peterson, 81][Silva, 85][Murata, 89][Gomes, 91][David et al., 92][Zurawsky et al., 94]. Em [Murata, 89] as propriedades das RdP são classificadas em comportamentais ou estruturais conforme sejam ou não dependentes da marcação inicial.

Dentro das propriedades comportamentais temos [Murata, 89: 547-50]: a alcançabilidade, a limitabilidade, a vivacidade, a reversabilidade, a cobertura, a persistência, a distância sincrónica e a justiça.

Dentro das propriedades estruturais refiram-se como de especial interesse [Murata, 89: 566-9]: vivacidade estrutural, controlabilidade, limitabilidade estrutural, conservabilidade, repetibilidade, consistência, justiça-B estrutural.

---

[18] Esta técnica foi sugerida por Henrik Hulgaard (henrik@cs.washington.edu) em resposta a uma pergunta de outro leitor da petri net mailing-list.

1.6 Métodos de Análise

Uma possível classificação dos métodos de análise de RdP considera a existência de três grupos [Murata, 89: 550]:

1. Análise por enumeração.
2. Técnicas de redução ou decomposição (transformações).
3. Matriz de incidência e equação de estado.

É ainda possível considerar um quarto tipo de análise de RdP: a simulação da RdP [Silva, 85]. Esta não permite obter garantias quanto a uma determinada propriedade do sistema modelado, mas por vezes pode ser a única técnica aplicável, dadas as limitações das restantes. Nomeadamente em RdP de alto-nível (vide §1.7.1) em que as restantes técnicas de análise ainda se encontram relativamente pouco desenvolvidas, a simulação apresenta uma grande importância.

A análise por enumeração baseia-se na construção de um grafo que representa todas as marcações que a RdP pode alcançar. Cada nó corresponde a uma marcação e cada arco corresponde ao disparo de um conjunto não vazio de transições, também denominado *passo*. Se a RdP é limitada, é possível construir este tipo de grafo e nesse caso ele denomina-se grafo de ocorrências [Jensen, 92]. Caso a RdP não seja limitada, o grafo de ocorrências é infinito. Nesse caso ainda é possível construir um grafo que se denomina *grafo de cobertura*[19]. Em [Murata, 89] e [David et al., 92: 36-7] encontra-se detalhado o algoritmo de construção da árvore, ou grafo, de cobertura.

Conforme notado em [Peterson, 77: 240] muitas questões podem ser reduzidas ao problema da alcançabilidade. No entanto, apesar das suas óbvias potencialidades e aplicabilidade[20], este método é muitas vezes dificilmente aplicável dada a sua natureza fortemente combinatória [Silva, 85: 101] com a consequente "explosão" de estados.

A análise por redução (ou transformação) consiste na obtenção de uma RdP mais simples mas que mantem as propriedades que se pretendem analisar. Este tipo de análise pode revelar-se muito útil

---

[19] Evitou-se, propositadamente, a designação "árvore de alcançabilidade" (*reachability tree*) que apresenta pelo menos dois significados distintos na literatura. Em [Murata, 89] é lhe atribuído um significado idêntico ao de grafo de alcançabilidade (igual a grafo de ocorrências [Jensen, 92]) embora utilizando representações gráficas distintas. Já em [Silva, 85] e [Peterson, 77][Peterson, 81], árvore de alcançabilidade tem o significado de árvore de cobertura de [Murata, 89] e [David et al., 92: 34-7]..

[20] É aplicável a todas as classes de RdP [Murata, 89: 550].

mas apenas é aplicável a alguns tipos especiais de RdP ou em algumas situações particulares [Murata, 89: 550]. Em [Murata, 89: 553] apresentam-se, de forma resumida, as técnicas de redução mais simples. Em [Silva, 85: 110-28] e [David et al., 92: 46-60] é feita uma apresentação mais detalhada das principais técnicas de análise por redução.

A matriz de incidências e a equação de estados constituem a tentativa de utilização de álgebra linear na análise de RdP. Tal como as técnicas de análise por redução, estes métodos não são aplicáveis a muitos tipos de RdP. Nas RdP onde é possível a sua aplicação, nomeadamente os grafos marcados [Silva, 85: 101], a sua utilização permite, em alguns tipos de análise, evitar as técnicas de enumeração. Em [Murata, 89] encontra-se uma boa introdução a este tipo de técnicas de análise.

1.7 Algumas classes de Redes de Petri

Desde o trabalho seminal de Petri têm surgido muitas e diversas variantes ao seu modelo de Redes de Petri. Pode-se afirmar que a maior parte destas variantes nasceu da necessidade de adaptação das RdP ordinárias à especificidade da aplicação para as quais a sua utilização era desejada. O apêndice A contem uma lista de alguns dos tipos de RdP mais frequentemente referidos na literatura.

Conforme referido em [Morasca et al., 91], o modelo original das RdP falha na representação de duas importantes características: aspectos funcionais complexos, tais como, condições que determinam o fluxo de controlo, e os aspectos de temporização. Para enfrentar estas duas limitações duas classes de extensões às RdP foram desenvolvidas: as RdP de alto-nível e as RdP temporizadas. A classe de RdP desenvolvida no âmbito deste trabalho, apresenta ambas as características: é de alto-nível e temporizada. Por essa razão, estes dois tipos de extensão são discutidos seguidamente.

1.7.1 Redes de Petri de alto nível

A característica fundamental que distingue as RdP de alto nível (RdP-AN) das RdP e que as qualifica como de alto nível, é a possibilidade de as marcas não serem iguais entre si, correspondendo a elementos de um domínio. Desta forma, as marcas podem conter muito mais informação. Esta já não se limita à simples presença ou não da marca (ou marcas) num determinado lugar, podendo conter dados relativos à sua caracterização como indivíduo distinto dos restantes.

Tal permite uma compactação (eventualmente muito significativa) da RdP, como veremos no exemplo típico do problema dos filósofos.

Dado as marcas não serem necessariamente iguais, torna-se necessário anotar a rede, designadamente os arcos e transições. As transições surgem como modificadores das marcas, podendo gerar marcas de domínios distintos dos das marcas presentes nos lugares de entrada.

Desta forma a complexidade da rede encontra-se dividida: parte é representada pela própria estrutura da rede - tal como sucede nas RdP ordinárias - e outra parte é representada pelas inscrições da rede. É importante notar que estas redes podem ser traduzidas para RdP de baixo nível, visto apresentarem o mesmo poder de modelação.

Numa rede de alto-nível em que cada lugar apenas possa conter, no máximo, uma marca podemos pensar nos lugares como variáveis (*valores-esquerdos*[21]). e nas marcas como valores de variáveis (*valores-direitos*). Este tipo de redes surge como uma generalização das RdP seguras ou binárias. Cada lugar já não corresponde necessariamente a uma variável *Booleana* (presença ou ausência de uma marca) mas pode corresponder a qualquer tipo de variável.

Se cada lugar poder conter mais do que uma marca (caso geral) podemos ver o lugar como uma estrutura de dados que contem um ou mais valores de um determinado tipo. Esta estrutura de dados é geralmente vista como um *multiconjunto*. O multiconjunto é a estrutura de dados que associada ao lugar, menos o afasta do lugar da RdP ordinária. Aí não existem diferenças entre as marcas pelo que não faz sentido falar em estruturas de dados associadas que devolvam diferentes valores consoante os critérios utilizados. Já nas RdP-AN podemos impor variadíssimos critérios resultantes das diferentes estruturas de dados que podemos associar a cada lugar. Por exemplo, alguns trabalhos consideram lugares *FIFO*[22] ou *LIFO*[23] [Pezzé, 94]. Outras estruturas podem ser utilizadas mas o multiconjunto é, de facto, aquela que permite a generalização natural dos lugares das RdP

---

[21] Do inglês *left value* ou *l-value*, endereço de memória. O nome deriva da operação de afectação. Aí o endereço corresponde à variável do lado esquerdo e o valor à variável ou constante do lado direito. Por exemplo, na afectação $A = B$, B representa um valor-direito (em inglês: *right-value* ou *r-value*) presente na posição de memória correspondente à variável B e que será colocado no endereço designado pela variável A. O r-*value* corresponde ao valor presente no endereço de memória de uma dada variável [Sethi, 90].

[22] Do Inglês: *First In First Out*, estrutura em fila (*Queue*), primeiro a entrar é o primeiro a sair.

[23] Do Inglês: *Last In First Out*, estrutura em pilha (*Stack*), último a entrar é o último a sair.

ordinárias e é a estrutura de dados em que se baseiam os dois principais tipos de RdP-AN: as RdP coloridas e as RdP Predicado-Transição. Os arcos de saída correspondem, nessas redes, às funções de adição de dados na estrutura e os arcos de entrada correspondem às funções de subtracção de dados da estrutura. É importante notar que não existe uma correspondência entre um tipo arco e uma função de modificação do valor de um campo da estrutura (uma marca). Esse efeito é conseguido através da conjugação das duas acções referidas: um arco de entrada retira a marca e um arco de saída adiciona uma nova marca.

Nas RdP-AN a cada lugar associamos um domínio ou tipo. A definição deste tem, como já referido, uma parte respeitante aos dados que pode conter. Por exemplo: um dado lugar pode ser responsável pela representação do conjunto dos livros requisitados numa biblioteca caso em que o seu tipo conterá um campo para o nome do livro, outro para o autor, outro para a editora, outra para a data de requisição, etc.. Se à definição do tipo adicionarmos as expressões dos arcos de entrada e de saída de todos os lugares desse tipo obteremos uma definição de tipo que contem quer a parte estática (reserva de memória para os seus dados) quer a parte de dinâmica de alteração desses dados. No exemplo apresentado uma função possível seria a de modificação da data de requisição do livro, a alterar sempre que o livro fosse de novo requisitado. Esta definição de um tipo corresponde ao que usualmente se denomina por tipo de dados abstracto. A definição dos tipos das variáveis como tipos de dados abstractos constitui uma das características da programação orientada por objectos [Meyer, 88][Booch, 94] tornando-a adequada para a implementação de RdP de alto-nível. A linguagem C++ permite a definição de tipos de variáveis que obedecem à definição apresentada em [Meyer, 88]. Como se verá nos capítulos 3 e 4, essa capacidade é largamente explorada no sistema desenvolvido.

O facto das marcas poderem representar variáveis de um tipo de dados abstracto, permite que através da utilização de marcas persistentes[24] se consigam especificar lugares que se comportam como essas estruturas[25]. Para tal basta que o lugar contenha uma marca cujo tipo corresponda à

---

[24] Numa RdP, considera-se uma marca persistente quando esta nunca *sai* do lugar, ou seja, para qualquer passo da rede, para cada arco de entrada existe um arco de saída que repõe a marca. No caso das RdP-AN podemos considerar uma definição mais lata em que a marca pode ser modificada. Por exemplo uma marca (de alto-nível) persistente, constituída por um identificador e uma fila de espera pode manter o identificador embora modificando o estado da sua fila de espera.

[25] Estas estruturas têm de ser limitadas caso se pretenda manter a possibilidade de traduçao da rede para uma RdP de baixo-nível.

estrutura de dados pretendida para o lugar. A Figura 1.5 mostra a especificação de um lugar *FIFO* utilizando este método. Para as inscrições da rede, utilizou-se um pseudocódigo muito próximo da linguagem C++ e da linguagem CpPNeTS-DL (ver §3.2) desenvolvida no âmbito desta dissertação. Os tipos correspondem a classes (directiva *Class*) e os lugares são declarados utilizando a directiva *Place*.

A maior parte do trabalho prático e teórico na área das RdP de alto-nível tem-se centrado ou nas RdP Predicado-Transição[26] [Genrich, 86] (RdPPr-Tr), ou nas RdP coloridas27 [Jensen, 94] (RdPCol). Conforme notado em [Jensen, et al., 91], os dois modelos são muito semelhantes, sendo mais correcto vê-los como dois dialectos, superficialmente diferentes, de uma mesma linguagem.

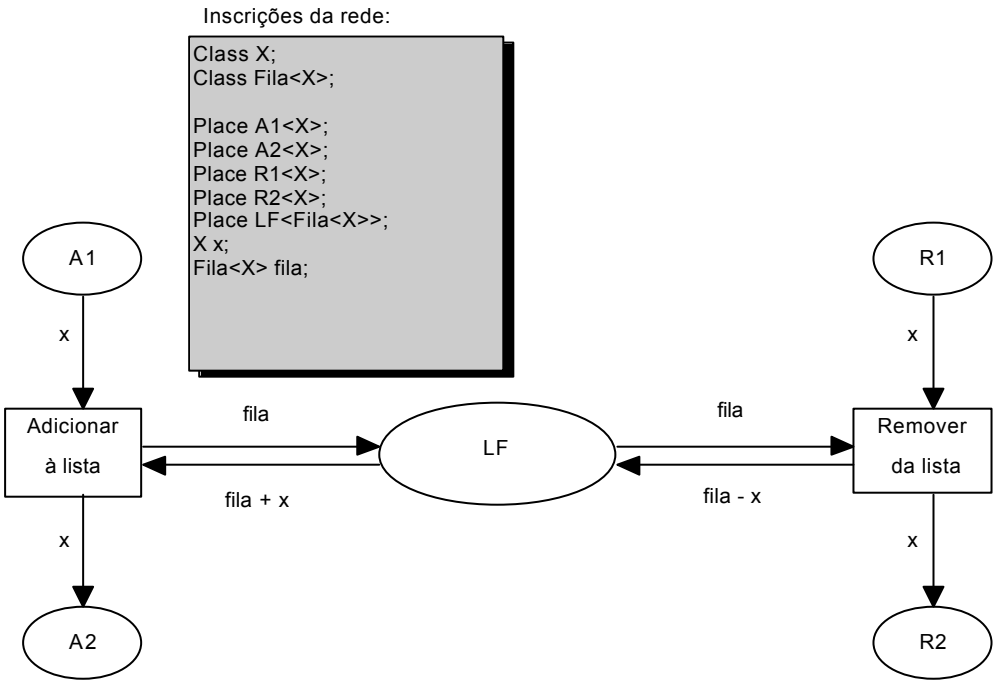


Figura 1.5 Especificação de um lugar *FIFO*, com base num lugar contendo um multiconjunto de marcas, numa RdP de alto-nível.

Outra classe de RdP de alto nível é a das Redes de Petri com marcas individuais (RdPcMI) [Reisig, 92]. Como se verá, o nível de abstracção permitido por esta classe de redes não é tão elevado como o das já referidas RdPCol e RdPPr-Tr.

---

[26] Em inglês: *Predicate-transition nets* ou *PrT-nets*.

[27] Em inglês: Coloured Petri Nets ou *CP-Nets*.

Em seguida faz-se uma apresentação das três classe de RdP de alto-nível. É dada uma maior ênfase às RdPCol, visto estarem na base do tipo de RdP desenvolvido para suporte ao sistema realizado.

Redes de Petri Predicado-Transição

As Redes de Petri Predicado-Transição modelam um sistema dinâmico em termos de um conjunto de indivíduos estruturado por um conjunto de funções e relações. As RdPPr-Tr permitem modelar as propriedades desses indivíduos e as relações entre eles, bem como as respectivas modificações. Intuitivamente, uma RdPPr-Tr permite dinamizar uma estrutura relacional R. Esta estrutura é um tuplo constituída por um domínio finito D de indivíduos, um conjunto de funções e um conjunto de relações. Podemos utilizar uma linguagem de primeira ordem para nos referirmos a uma estrutura relacional R se associarmos a cada operador uma função de R e a cada predicado uma relação de R. O alfabeto não-lógico dessa linguagem L compreende um conjunto de símbolos de operadores Ω, um conjunto de símbolos de predicados Π, e um conjunto de símbolos de variáveis V. Há também que definir um modelo $M = (D, W)$ para L, em que D é o conjunto de indivíduos já referido e W a função de interpretação. Também utilizado é o conjunto de predicados variáveis: Ψ.

A cada lugar corresponde um predicado variável: $\psi^n \in \Psi$. É devido a esta identificação dos lugares com os predicados variáveis que este tipo de redes se denomina Predicado-Transição, por oposição a Lugar-Transição[28]. Cada lugar encontra-se marcado por um conjunto (interpretação forte) ou multiconjunto (interpretação fraca) de tuplos de indivíduos de aridade n. Este tuplos representam as interpretações actuais dos predicados associados. Para uma marcação de um dado lugar $p \in P$ anotado por ψ^n., dizemos que o predicado variável ψ^n é verdadeiro segundo o vínculo $\alpha = [v_1 \leftarrow d_1,..., v_n \leftarrow d_n]$, com $v_i \in \mathrm{Var}(\psi^n)$ e $d_i \in D$, se e só se $(d_1,...,d_n) \in M(p)$.

O disparo de uma transição (ocorrência de um evento), muda as interpretações dos lugares de entrada para novas interpretações nos seus lugares de saída. Este efeito é expresso através de somas simbólicas de tuplos de termos da linguagem L que constituem as inscrições nos arcos da RdPPr-Tr. Estas somas simbólicas de tuplos de termos indicam a posição e o nome para os valores contidos nos tuplos dos lugares de entrada, permitindo dessa forma seleccionar quais os campos

[28] A designação RdP Lugar-Transição corresponde à RdP generalizada (vide Apêndice A).

dos tuplos das relações que serão considerados no disparo da transição. Idênticas inscrições nos arcos de saída especificam os tuplos (marcas) a gerar nos lugares de saída.

Cada transição tem associado um *selector*[29]. Este é uma fórmula de *L* que não contem predicados variáveis. Um disparo da transição implica um vínculo, para cada uma das variáveis na sua *vizinhança*, que torne o selector verdadeiro.

Uma apresentação mais completa e detalhada das RdPPr-Tr, assim como das condições de disparo das suas transições encontram-se para alem do âmbito desta dissertação. Como tal recomenda-se a referência principal [Genrich, 86] ou, para uma definição também formal mas mais condensada, a secção 1.1 de [Lindqvist, 90].

Redes de Petri Coloridas

As Redes de Petri coloridas são, provavelmente, as RdP-AN que mais interesse têm levantado. O salto que se dá ao passar das RdP para as RdPCol é, mais do que em qualquer outro tipo de RdP de alto nível, extremamente semelhante ao que se dá ao passar da programação em linguagens *Assembly* para linguagens de alto-nível como PASCAL ou C [Kernigham et al., 88]. A utilização de estruturas de dados complexas em vez de *bits* tem o seu paralelo nas RdP, na utilização de marcas complexas em substituição das marcas simples das RdP ordinárias.

O, já citado, exemplo dos filósofos permite uma demonstração do poder de compactação da rede oferecido pelas RdPCol. Em §1.7.3 veremos como a introdução de uma representação hierárquica nas RdP permite a ocultação de sub-redes que surgem repetidas, através do recurso a instâncias de uma RdP que surgem na rede original como macrolugares ou macrotransições (§1.7.3). As RdPCol também podem facilmente suportar uma representação hierárquica com todas as vantagens que esta apresenta, mas a sua principal vantagem reside no elevado poder de compactação da rede, resultante, não de uma representação hierárquica mas sim, do facto das marcas corresponderem a variáveis de tipos cuja complexidade apenas se encontra dependente da linguagem utilizada para os definir, normalmente a mesma que é utilizada para todas as inscrições da rede. Na Figura 1.6

[29] Funcionalmente idêntico à *guarda* nas RdP coloridas. Em [Genrich, 89] é já adoptado o termo *guarda* para designar o *selector*.

ilustra-se a transformação de uma RdP ordinária numa RdPCol, utilizando o bem conhecido problema dos filósofos [Ben-Ari, 82].

A transformação consiste fundamentalmente na substituição de conjuntos de lugares por um só lugar "colorido", ou seja, contendo marcas coloridas. Essas marcas coloridas, permitem a representação de cada um desses lugares através de valores (neste caso inteiros) distintos. Esta fusão de lugares obriga necessariamente a uma fusão dos respectivos arcos. Esta é conseguida associando funções aos arcos que permitam determinar quais as marcas a subtrair ou adicionar aos lugares. É o que acontece com a função Garfos() na Figura 1.6b). É de salientar que a redução de complexidade obtida através da utilização da rede colorida (Figura 1.6 b)) será tanto mais significativa, quanto o maior for a quantidade de filósofos. Para se obter uma representação de uma mesa com mais filósofos e garfos, seria apenas necessário alterar a marcação inicial dos lugares `Pensando` e `Garfos disponíveis` e a função `Garfos`. Enquanto que em a) a própria rede cresceria, na quantidade de lugares e arcos, na razão directa do crescimento do número de filósofos e garfos.

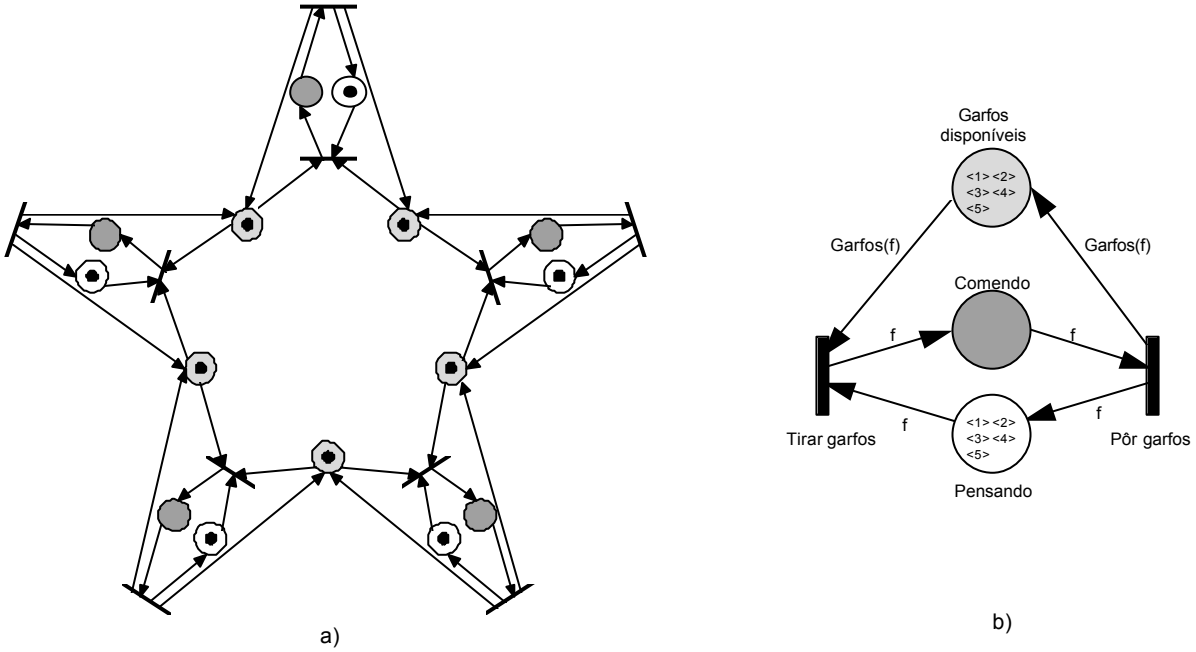


Figura 1.6 O problema dos filósofos: a) utilizando uma RdP ordinária; b) utilizando uma RdPCol. Os lugares em a) apresentam colorações distintas para evidenciar a sua "fusão" no lugar com idêntica coloração em b).

A selecção das marcas a subtrair ou adicionar aos lugares é suportada por um conjunto de variáveis, também chamadas variáveis de transição visto serem sempre utilizadas no contexto do disparo de uma transição. Como o cálculo do disparo de cada transição é feito em diferentes instantes, as

variáveis podem ser declaradas num escopo global (conhecidas portanto de todas as transições) sendo, nesse caso, utilizadas por mais do que uma transição. A necessidade de existência de variáveis advém, tal como as expressões dos arcos, do facto, das marcas não serem todas iguais. A necessidade da existência de variáveis é uma das consequências directas da existência de marcas com valores associados. Esses valores têm de ser conhecidos durante o processo de disparo de uma transição pelo que as variáveis servem de depósito das marcas escolhidas. Podem então ser lidas com vista à avaliação da guarda e da determinação dos valores das marcas a gerar. Nas RdP ordinárias o facto de todas as marcas serem iguais, implica que o único dado importante no disparo da transição é a presença ou não da quantidade de marcas necessária nos arcos de entrada. As marcas geradas são aí apenas determinadas pelos pesos (RdP generalizada) dos arcos de saída. Não existe a noção de *valor* de cada marca.

Conforme referido, as variáveis na RdPCol servem para guardar os valores das marcas, dos arcos de entrada, seleccionadas. O par *(variável, valor selecionado)*, denominar-se-á *vínculo*[30] e representar-se-á por: <*variável*←*valor*>. Por exemplo, se a variável *x*, do tipo inteiro, tem o valor *3* atribuído então o vínculo correspondente será <*x*←*3*>.

Dado que as marcas não são necessariamente iguais entre si, é extremamente útil a possibilidade de impor restrições ao disparo de uma transição com base nos valores das suas marcas. Para tal define-se uma função *Booleana* que permite testar os valores das marcas nos lugares de entrada, que à partida poderiam ser utilizadas no disparo de uma transição. A cada transição pode então associar-se uma função desse tipo denominada *guarda*.

Para uma definição formal das RdPCol e seu comportamento dinâmico recomenda-se a leitura de [Jensen, 92]. Para uma análise formal mais detalhada de alguns aspectos, uma excelente continuação é [Jensen, 94].

[30] Em inglês: *binding*.

Redes de Petri com marcas individuais

As redes de Petri com marcas individuais [Reisig, 83][31] [Reisig, 92: 55-8] são redes de alto-nível que conseguem manter uma grande semelhança com as redes lugar-transição.

> "A ideia da introdução de marcas individuais não é, de forma alguma, inovadora.(...)Portanto, a introdução de mais um modelo merece alguma motivação. O nosso objectivo principal é um modelo simples que permita uma descrição clara e precisa do seu comportamento e métodos de análise. A intuição por detrás de todas as construções deve ser tão transparente e óbvia como para as redes lugar-transição." [Reisig, 83: 230].

Em [Reisig, 92], onde as RdP são introduzidas de uma forma muito pedagógica, o autor começa por apresentar as RdP condição-evento avançando gradualmente para as RdP lugar-transição e para as RdP com marcas individuais. O caminho aí apresentado é muito "natural" constituindo uma excelente introdução à motivação para as RdP de alto-nível.

As RdP com marcas individuais apresentam muitas semelhanças com as redes de Petri coloridas, de tal forma que podem parecer uma versão simplificada destas. As diferenças fundamentais resumem-se a:

- Todos os lugares contêm apenas um tipo de marcas. Essas marcas são multiconjuntos de elementos de um domínio D, domínio esse que inclui a constante especial "marca preta" que não necessita ser indicada nas expressões dos arcos [Reisig, 92: 55].
- As expressões dos arcos não contêm chamadas de funções.
- As guardas (denominadas *condições*) não contêm chamadas de funções.

1.7.2 Redes de Petri com temporizações associadas

> *Pois o tempo é a distancia mais longa entre dois lugares.*
>
> Tennessee Williams

Existem várias extensões às RdP com vista a considerar a modelação do tempo. Tal é necessário para a avaliação da *performance* de sistemas e no escalonamento de sistemas dinâmicos. Para tal associam-se tempos de atraso aos lugares e/ou transições. Podem ser qualificadas de

[31] As RdP com marcas individuais são aí descritas com a designação: "Relation-Net Model".

determinísticas, se os tempos são dados de forma determinística, ou probabilísticas, se os tempos são determinados de forma probabilística. Estas são extremamente úteis quando se pretendem estudar desempenhos médios dos sistemas modelados. Dado que o sistema desenvolvido tem como objectivo, a geração de código para execução da RdP a partir da modelação de um sistema de tempo real, apenas se descrevem sumariamente os principais tipos de RdP temporizadas determinísticas.

Existem dois modelos básicos de RdP que permitem a modelação de temporizações:

1. As RdP temporizadas[32] [Ramchandani, 74] em [Zuberek, 91].
2. As RdP com temporizações[33] [Merlin et al., 76] em [Berthomieu et al., 91].

As RdP temporizadas são RdP às quais se associam tempos de duração para o disparo das transições, ou tempos de indisponibilidade das marcas presentes nos lugares. No primeiro caso, as RdP temporizadas são denominadas *temporizadas-T*, e no segundo caso *temporizadas-L*. Uma RdP temporizada é, por convenção, temporizada-T [Zurawsky et al., 94]. Em ambos os tipos de temporização, os tempos associados podem ser determinísticos ou probabilísticos pelo que as RdP temporizadas podem ser qualificadas de determinísticas ou de probabilísticas [Molloy et al., 82][34]. Estas não podem ser resolvidas analiticamente, para casos gerais, se os atrasos não forem estocásticos e distribuídos exponencialmente. Quando tal se verifica as redes são denominadas estocásticas. Quando também se admitem transições com disparo instantâneo, denominam-se estocásticas generalizadas [Marsan et al., 84][34] [Marsan et al., 86] [34]. Em [Murata, 89: 570-2] pode encontrar-se uma breve introdução às RdP estocásticas e estocásticas generalizadas, na modelação do desempenho de sistemas.

Nas RdP temporizadas-T, a regra de disparo é modificada em dois aspectos:

1. É necessário considerar o tempo que a transição demora a disparar;
2. A transição tem obrigatoriamente de disparar logo que se encontre habilitada.

Tal significa que, após a remoção das marcas dos lugares de entrada, as marcas só serão depositadas nos lugares de saída após decorrido o tempo associado à transição. Assim sendo,

---

[32] Denominadas *timed Petri nets* na bibliografia em língua inglesa.

[33] Denominadas *time Petri nets*, na bibliografia em língua inglesa.

[34] Conforme referido em [Zurawsky et al., 94].

qualquer descrição dos estados da RdP tem de ter em conta não apenas as marcações dos lugares mas também as marcas que se encontram "dentro" da transição. Isto significa que o conjunto de marcações (ou estados) da RdP temporizada pode ser muito diferente do conjunto de marcações (ou estados) que se alcançam com a RdP não temporizada. As marcas nos lugares encontram-se sempre disponíveis para habilitar as transições.

Nas RdP temporizadas-L, a temporização é feita através dos lugares. Quando as marcas são depositadas num lugar temporizado, apenas ficam disponíveis para habilitar outras transições após decorrido o tempo associado ao lugar. O disparo das transições é instantâneo tal como nas RdP ordinárias.

Ambos os modelos são equivalentes, conforme já demonstrado em [Ramchandani, 74]. Dado um modelo temporizado-L, o modelo temporizado-T pode ser obtido por substituição de cada um dos lugares temporizados por uma rede constituída por um lugar, uma transição com temporização igual ao lugar substituído, e outro lugar. O procedimento é simétrico para passar de uma rede temporizada-T para uma temporizada-L, (vide Figura 1.7).

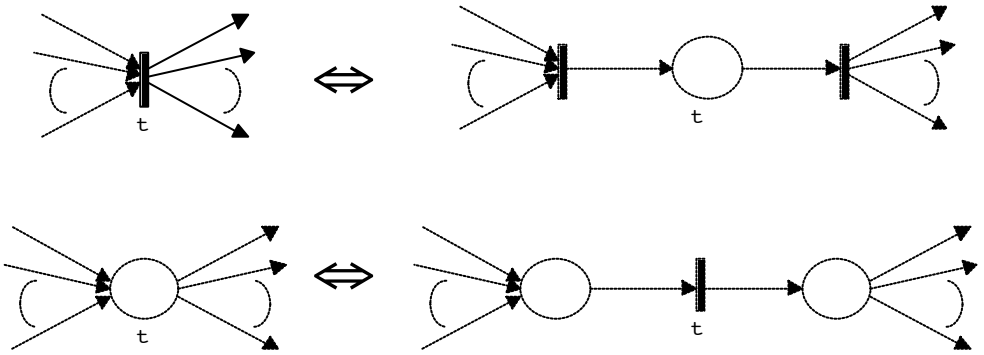


Figura 1.7 Equivalências entre RdP temporizadas-L e temporizadas-T.

As RdP com temporizações são mais gerais que as RdP temporizadas. Estas podem ser modeladas através de RdP com temporizações, mas o contrário não é possível [Berthomieu et al., 91]. Nas RdP com temporizações, cada transição está associada a um intervalo de tempo: $0 \le a \le b \le \infty$. O valor a do intervalo é um valor mínimo, contado a partir do instante em que a transição fica apta, que deve decorrer até a transição disparar. O valor b é o tempo máximo que a transição pode estar apta sem disparar. Por outras palavras cada transição tem associado um intervalo de tempo durante o qual tem necessariamente de disparar.

Em [Kumar et al., 94] é apresentado um modelo de RdP temporizadas nos lugares e nas transições, pelo que as temporizadas-L e temporizadas-T surgem como casos particulares desse modelo. Por outro lado, o tempo associado com um lugar ou uma transição não é já um simples atraso podendo ter associado um mecanismo de fila-de-espera arbitrário [Kumar et al., 94:1501-2].

Importa frisar que estes modelos de RdP apresentam uma semântica distinta da das RdP. As temporizações associadas às transições podem impedir ou forçar as transições a disparar, e as temporizações associadas aos lugares podem impedir a habilitação de transições.

Em [Jensen, 95:164-5], citam-se várias referência para algumas das publicações mais significativas relacionadas com modelos de RdP com temporizações associadas. Em [Zuberek, 91] e [Kumar et al., 94] encontram-se, também, bastantes referências sobre RdP temporizadas.

Redes de Petri de alto-nível temporizadas - as RdP Coloridas Temporizadas

No caso das RdP-AN torna-se possível considerar um atributo de cor do tipo inteiro positivo que corresponde a um instante de tempo. Foi esse o caminho seguido na definição das RdPCol temporizadas [Jensen, 95: 145-65]. Assim sendo, a temporização não está directamente associada aos lugares, o que implicaria uma igual temporização para todas as marcas neles presentes, mas sim às próprias marcas. Esta maior "precisão" na especificação do tempo é uma consequência natural da condensação de lugares que ocorre aquando da utilização de uma RdPCol. Por exemplo, as redes da Figura 1.6, mostram claramente que caso a RdPCol apenas permitisse a especificação de temporizações associadas aos lugares, se perderia a capacidade de especificar durações distintas para o "comer" e o "pensar" dos vários filósofos. Dessa forma as RdPCol temporizadas apresentariam um poder de modelação inferior ao das RdP temporizadas, o que obviamente se pretendeu evitar.

Nas RdP coloridas temporizadas [Jensen, 95:145-65], considera-se a existência de um tempo global sempre crescente. Só quando o tempo da marca é menor ou igual ao tempo global, é que esta pode ser retirada do lugar. Nessa situação a marca diz-se *pronta*. Um vínculo de transição diz-se habilitado pela cor quando se encontra habilitado como numa RdPCol não temporizada. Para estar apto (numa RdPCol temporizada), necessita não só de estar habilitado pela cor mas também "pronto". Tal significa que todas as marcas removidas pela aplicação do vínculo têm de apresentar

valores de tempo menores ou iguais que o tempo global.

A execução de uma RdPCol temporizada é semelhante à das redes interpretadas [David et al., 92] e funciona de forma semelhante à das filas de eventos de muitas linguagens de programação para a simulação de eventos discretos. O tempo global do sistema mantém-se enquanto houver vínculos habilitados pela cor e prontos. Quando já não se puderem executar mais vínculos, o tempo global é avançado até o sistema se encontrar num novo "estado" em que existam vínculos que possam ser executados [Jensen, 95:146].

Os tempos das marcas são impostos pelas expressões dos arcos de entrada dos lugares, de forma semelhante ao que é feito com os restantes atributos da marca. Desta forma, é fácil afectar os tempos utilizando funções que implementam distribuições estatísticas complexas.

Os arcos de saída dos lugares permitem a extracção de marcas uma determinada quantidade de tempo antes destas se encontrarem prontas. Tal corresponde a utilizar uma definição mais lata para o operador ≤ entre multiconjuntos. De facto, se se considerasse o operador ≤ utilizado nas RdPCol não-temporizadas, seria necessário que todas as marcas resultantes da avaliação da expressão do arco (de saída do lugar), apresentassem exactamente o mesmo valor do atributo tempo que as marcas a subtrair do lugar. Tal condição foi considerada demasiado forte. A solução adoptada de utilizar um operador ≤ mais "fraco", torna possível, por exemplo, implementar um temporizador em que um arco de saída o pode interromper antes do tempo programado. Na Figura 1.8 apresenta-se a modelação desse temporizador que obriga a uma espera de 12 unidades de tempo para passar do estado A (marca no lugar p_1) para o estado B(marca no lugar p_3). Uma situação de alarme pode interromper a espera, forçando a evolução para o estado B.

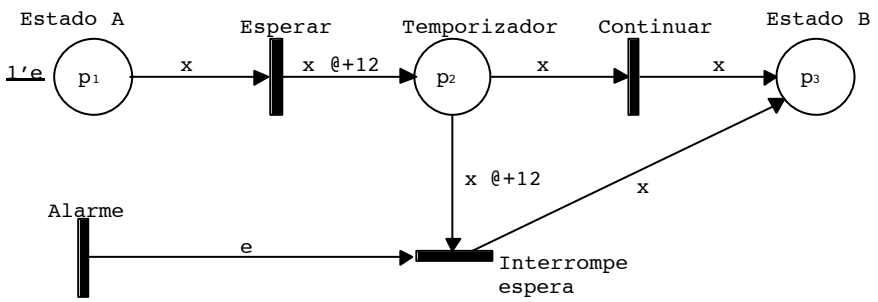


Figura 1.8 Modelo de temporizador utilizando uma RdPCol temporizada. É utilizada a notação de [Jensen, 95].

Tal como as RdP temporizadas e as RdP com temporizações, as RdPCol temporizadas foram definidas tendo em vista o estudo do desempenho de sistemas, como, por exemplo, o tempo máximo de execução de determinadas actividades e o tempo médio de determinadas respostas. De resto, a análise de desempenho é uma das maiores e mais importantes sub-áreas de aplicação das RdP [Jensen, 95].

Do que ficou dito, é fácil verificar que é possível transformar uma RdPCol temporizada numa RdPCol não-temporizada, bastando para tal omitir os atributos tempo de todas as marcas. Comparando os respectivos grafos de ocorrências, verifica-se então, que o das RdPCol temporizadas é um sub-grafo do das RdPCol não-temporizadas. A temporização da RdPCol tem um efeito semelhante ao das outras restrições especificáveis no modelo (as guardas e expressões dos arcos): limita as possíveis evoluções da rede.

Em [Morasca et al., 91] é apresentado um modelo de RdP-AN temporizada (as ER-nets) que não pressupõe a existência de um relógio global e que com base em dois axiomas que asseguram a monoticidade do tempo, apresenta uma semântica idêntica à das RdP ordinárias. Não obstante tal facto, as temporizações numa ER-net continuam a impor restrições que necessariamente limitam as evoluções da rede.

1.7.3 Redes de Petri hierárquicas

Uma RdP que modele um sistema não trivial, rapidamente se torna demasiado extensa para poder ser facilmente compreendida. Nestas situações o velho princípio "dividir para reinar", surge como a forma mais óbvia de reduzir a complexidade e, desse modo, manter a legibilidade do modelo.

A forma mais natural e intuitiva de decompor uma RdP é a de considerar a existência de sub-redes que se comportam como lugares ou como transições. Desta forma, uma RdP passa a incluir mais dois conjuntos de elementos: os macrolugares e as macrotransições, conforme se tratem de sub-redes que se comportem como lugares ou como transições, respectivamente. Por exemplo, a rede da Figura 1.6a) pode ser representada através de uma rede hierárquica em que cada filósofo é modelado por uma instância de uma macrotransição (Figura 1.9).

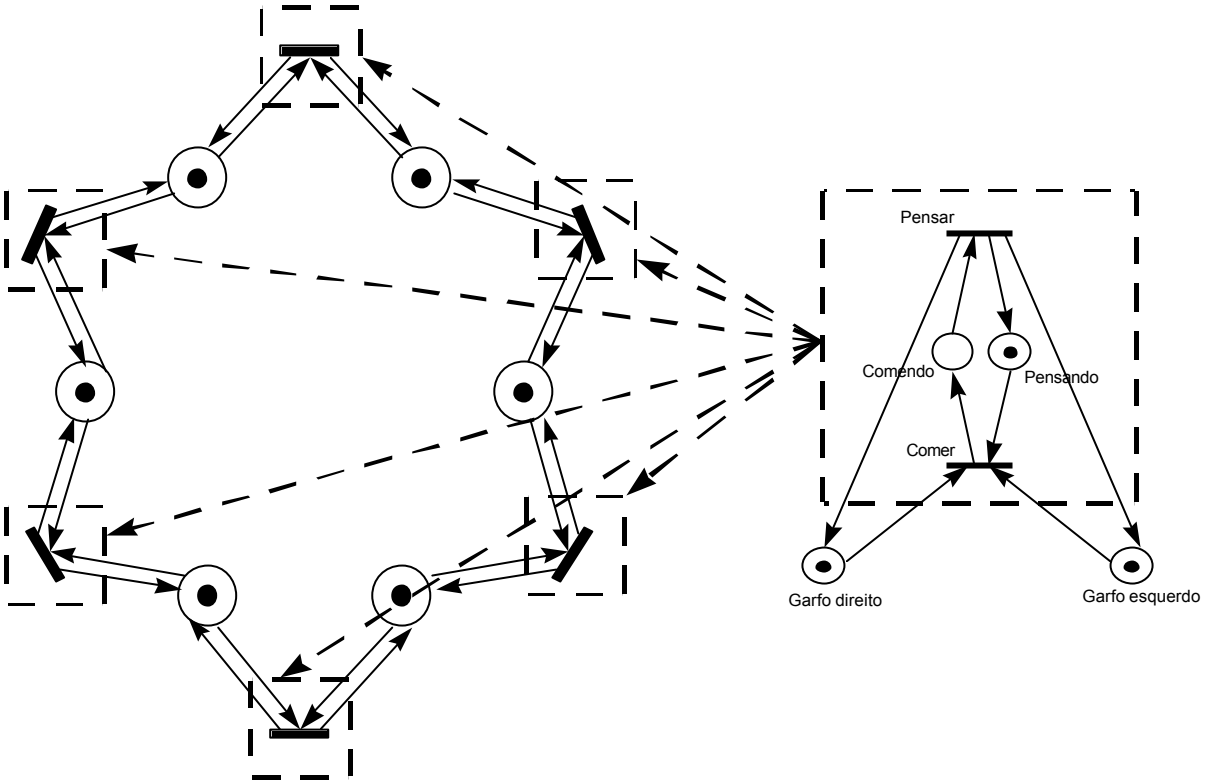


Figura 1.9 Problema dos filósofos modelado por uma RdP com macrotransições. cada macrotransição (à direita na figura) representa um filósofo. A rede é equivalente às da Figura 1.6.

Numa macrotransição ou num macrolugar, os nós (lugares ou transições) que contactam com o exterior da macro (como acontece com as transições *Comer* e *Pensar* da Figura 1.9), denominam-se portos de entrada, de saída ou de entrada-saída conforme se encontrem ligados ao exterior por arcos de entrada, saída ou ambos, respectivamente. Daqui podemos concluir que os portos de uma macrotransição são necessariamente transições, e os portos de um macrolugar são necessariamente lugares. No entanto, se definirmos dentro de uma macrotransição lugares "fantasma" de interface, estes podem ser considerados como os lugares porto da transição. Estes lugares "fantasma" não são verdadeiros lugares pois são fundidos com os verdadeiros lugares da rede que inclui a macrotransição. Este tipo de lugares é utilizado nas RdPCol e também no protótipo desenvolvido (vide §3.3.6).

Também aqui é possível traçar um paralelo com as linguagens de programação: enquanto as RdP-AN e, em particular as RdPCol, têm a sua correspondência nas linguagens de alto-nível, as

redes hierárquicas correspondem a uma linguagem *assembly* que disponibilize *macros*[35]. Assim temos:

Redes de Petri	Linguagens de Programação
RdP ordinárias	Linguagem *Assembly*
RdP hierárquicas	Linguagem *Assembly* com *macros*.
RdP alto nível	Linguagem de alto nível

Quadro 1.3 Redes de Petri *versus* linguagens de programação.

Redes de Petri coloridas hierárquicas

Embora teoricamente interessantes e potencialmente úteis as RdP hierárquicas não resolvem eficientemente o problema do crescimento excessivo do número de nós da rede, quando se pretende modelar um sistema não trivial. Como vimos, as RdP-AN e designadamente as RdPCol, apesar de teoricamente possuírem o mesmo poder de modelação das RdP ordinárias, na prática apresentam uma muito maior capacidade de modelação do que as RdP ordinárias, mesmo considerando construções hierárquicas. Conforme já notado em [Huber et al., 90], a diferença é idêntica à que existe entre programar utilizando linguagem *assembly*, e programar utilizando uma linguagem com tipos de dados estruturados.

1.8 Redes de Petri não-autónomas

As RdP denominam-se autónomas quando a sua evolução apenas depende de condicionalismos resultantes da sua estrutura e marcação. As RdP com temporizações associadas são um exemplo de RdP não-autónomas, dado o seu comportamento depender, também, de condições impostas pelas temporizações associadas. Outra forma de redes não-autónomas resulta de considerarmos o disparo, de uma ou mais transições, dependente, não apenas da marcação dos seus lugares de entrada, mas também de uma condição externa associada. Esta vai permitir a sincronização da evolução da rede com eventos externos. Estas RdP denominam-se *sincronizadas*.

O sistema desenvolvido permite a modelação do controlo de sistemas a eventos discretos, razão pela qual a RdP que lhe serve de suporte necessita sincronizar-se com o exterior.

---

[35] Porção de código com um dado nome. Esse código pode ser invocado multiplas vezes num mesmo programa comportando-se como um sub-programa. A invocação do código (correspondente ao nome deste) é substituída textualmente antes do texto ser compilado.

1.9 Conclusão

As RdP oferecem um suporte prometedor para a modelação, análise e simulação de sistemas a eventos discretos. O baixo-nível do modelo seminal das RdP torna extremamente difícil a sua aplicação para sistemas não triviais. Como tal várias RdP de alto-nível foram sendo desenvolvidas e estudadas. Destas, foram referidas as três mais frequentemente citadas. Todas apresentam a vantagem de permitirem especificações muito mais compactas tornando dessa forma posível a sua aplicação a sistemas muito mais complexos. Outra limitação do modelo seminal, é a impossibilidade de modelação do tempo. Tal é necessário para a modelação de sistemas com exigências de tempo-real. Em resposta a esta limitação várias forma de "RdP com tempo" foram surgindo. As duas mais significativas, RdP temporizadas e RdP com temporizações, foram apresentadas. Seguidamente, indicou-se como a decomposição hierárquica pode ser aplicada às RdP e como é ainda mais útil quando se utilizam RdP de alto-nível. Por fim, referiram-se as RdP sincronizadas com eventos externos.

No próximo capítulo apresenta-se uma nova classe de RdP de alto-nível e com temporizações associadas que permite a especificação da sua sincronização com eventos externos de forma a possibilitar a modelação pretendida.

Capítulo 2

O Sistema

O sistema CpPNeTS-S (vide Figura 2.1) é constituído por um conjunto de ferramentas que permitem a especificação e análise de sistemas modelados através de CpPNeTS (*C++ based Coloured Petri Nets with Time and Synchronisation*), uma nova classe de redes de Petri de alto-nível, herdeira da classe de RdP apresentada em [Gomes et al., 92]. Os constituintes fundamentais, deste sistema, são:

- Um tradutor que transforma uma especificação na linguagem CpPNeTS-DL §3.2 (*C++ based Coloured Petri Nets with Time and Synchronisation Description Language*) em código C++.
- Uma biblioteca de classes C++ para suporte às várias operações de análise das CpPNeTS que se pretendam implementar.

As classes C++, correspondentes à descrição da rede, geradas pelo tradutor, são compiladas e posteriormente ligadas à biblioteca (descrita no capítulo 4). Essa biblioteca suporta a representação da rede e oferece procedimentos que operam sobre essa representação.

Optou-se pela linguagem C++ para implementação do sistema. Quer o código gerado pelo tradutor, quer a biblioteca com o qual este é ligado, foram desenvolvidos utilizando essa linguagem. As próprias inscrições da rede são escritas na linguagem C++. Tal apresenta duas grandes vantagens: por um lado utiliza-se uma linguagem já existente evitando-se o desenvolvimento de outra linguagem, necessariamente menos testada e conhecida, por outro possibilita-se uma integração perfeita com a biblioteca, também ela desenvolvida utilizando a linguagem C++. No fim do presente capítulo, em §2.2, explicitam-se as razões desta opção.

Nos capítulos 3 e 4, serão apresentados os protótipos desenvolvidos respectivamente para o pré-processador e biblioteca.

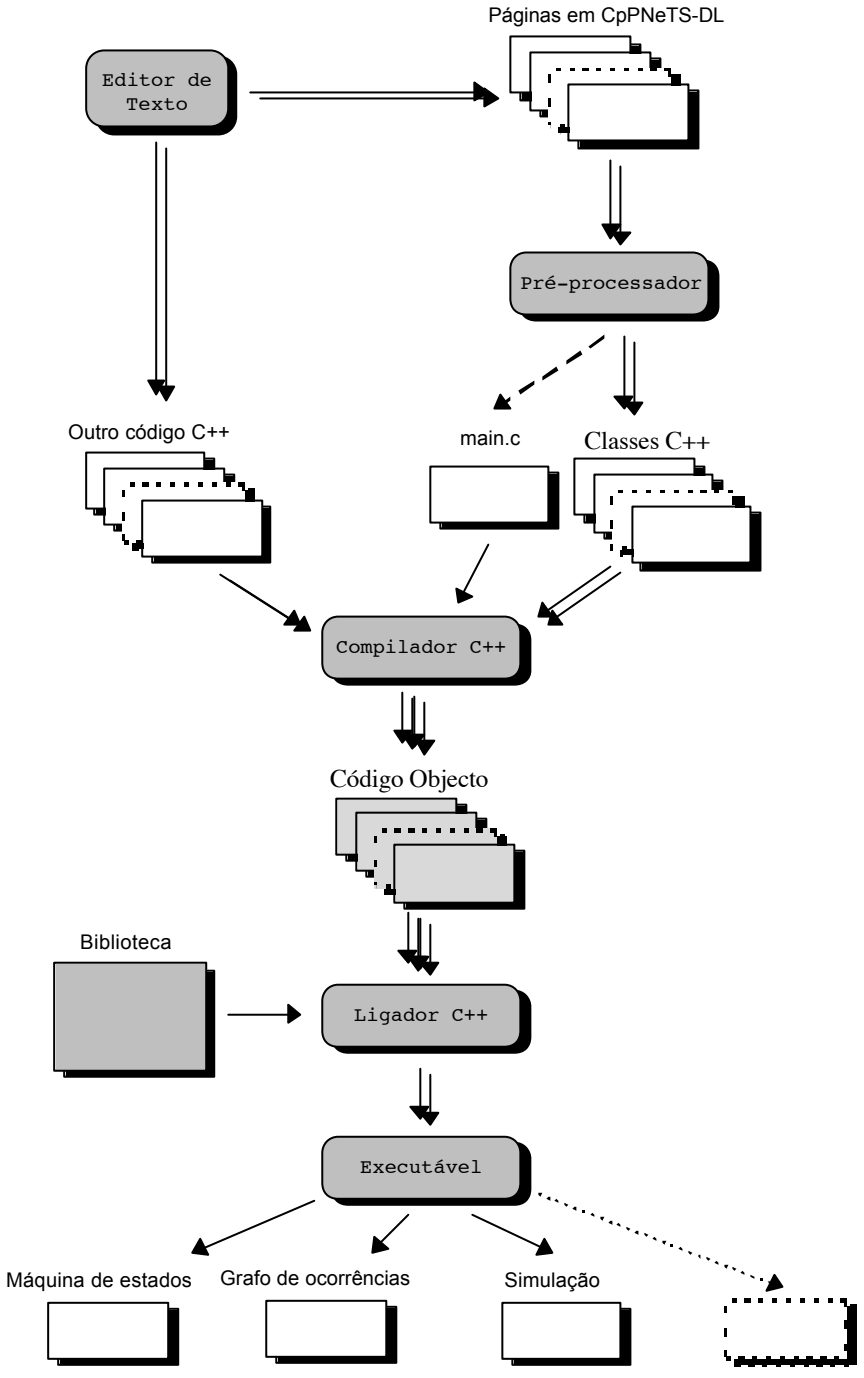


Figura 2.1 O sistema CpPNeTS-S.

No restante deste capítulo, apresenta-se uma nova classe de Redes de Petri Coloridas, as CpPNeTS, bem como, as motivações para a utilização da linguagem C++. O capítulo termina com um conjunto de breves descrições de alguns sistemas conhecidos da literatura que apresentam

semelhanças importantes com o sistema CpPNeTS-S.

2.1 Uma nova classe de Redes de Petri Coloridas

nimium ne crede colori

Virgílio in *Bucólicas*

Em [Gomes et al., 92], a especificação do controlador, é efectuada recorrendo a um modelo de RdP coloridas [Jensen, 92] a que se adicionaram capacidades de sincronização semelhantes às descritas em [David, 91]. Aí, uma transição apenas pode disparar se está apta quando a condição externa a ela associada é verdadeira. Esta condição externa, ou evento, é função das variáveis de entrada. As actuações externas são efectuadas recorrendo a listas de pares *(condição, acção)* associadas aos lugares. A condição é função das marcas e a acção actua variáveis de saída do sistema. A rede é temporizada nas marcas utilizando um atributo de cor especial denominado *duração*. Em cada ciclo, subtrai-se uma unidade a esse atributo. A marca apenas se encontra disponível quando o valor do atributo é zero. Dessa forma a temporização é independente para cada marca. Como métodos de realização são referidos dois: o directo, em que o executor utiliza directamente a estrutura e características da rede e o indirecto em que se obtém um árvore de acessibilidade estruturalmente semelhante a uma máquina de estados. O método directo é considerado muito exigente do ponto de vista de recursos computacionais, apresentado-se o método indirecto como mais interessante. À classe de RdP proposta, e na ausência de uma designação anterior, vamos chamar RdP-CST (redes de Petri coloridas, sincronizadas e temporizadas).

Tal como as RdP-CST, as CpPNeTS constituem uma generalização das RdPCol [Jensen, 92], com

vista a suportar a modelação de sistemas sincronizados por eventos externos, bem como, a especificação de acções externas. A especificação quer de eventos ou acções externos quer do tempo, é opcional. Tal significa que a rede a construir pode, ou não, ser autónoma [Silva, 95][David, 91][David et al., 92].

As CpPNeTS apresentam todas as características das RdP-CST, e acrescentam mais três:

1. Tal como as RdPCol-H, as CpPNeTS podem ser especificadas por um conjunto de páginas. Estas diferem no entanto da páginas apresentadas em [Jensen, 94: 91] no facto de

poderem também constituir macrolugares e não apenas macrotransições[36]. Por outro lado não englobam o conceito de fusão de lugares [Jensen, 94: 97].

2. Permitem a especificação de actuações externas associadas às transições e não apenas aos lugares.

3. As transições podem ter prioridades associadas, como forma do utilizador impor uma ordem de disparo.

Conforme já referido, a especificação hierárquica da rede é muito útil. É sempre possível encontrar a especificação não-hierárquica equivalente à especificação hierárquica. A Biblioteca implementada, embora mantendo a estrutura hierárquica da rede, oferece também uma vista "plana" que é utilizada para implementar os procedimentos de análise e simulação. Actualmente, a representação hierárquica apenas é utilizada aquando da especificação. Por esta razão, optou-se por apresentar uma definição formal das CpPNeTS que não engloba a representação hierárquica. Esta é discutida de um modo mais informal aquando da apresentação do tradutor *pnetcpp* (no capítulo 3) . Este programa traduz a especificação da rede na linguagem CpPNeTS-DL em código na linguagem C++.

As actuações associadas às transições podem ser função do vínculo da transição. Já as actuações associadas aos lugares são função das marcas presentes nos respectivos lugares e/ou marcação de quaisquer lugares.

As prioridades associadas às transições são dadas por números inteiros positivos e especificam a ordem de disparo das transições pertencentes a uma mesma contenda (vide §2.1.1).

2.1.1 Estrutura das CpPNeTS

A definição das CpPNeTS apresenta algumas diferenças face à definição das RdPCol apresentada em [Jensen, 92: 70]. Algumas destas diferenças resultam da opção de não abstrair totalmente o

[36] As macrotransições correspondem às *substitution transitions* [Huber et al., 90][Jensen, 94], mas os macrolugares não correspondem aos *substitution places*. Os macrolugares são apenas sub-redes sem lugares nem transições porto (vide §3.3.6).

facto de a linguagem C++ constituir a linguagem utilizada para as inscrições da rede. Assim temos que as expressões dos arcos e as guardas correspondem a funções C++.

Com vista a simplificar a apresentação das CpPNeTS considerou-se a existência dos seguintes conjuntos, funções e notações:

a) O conjunto F das funções na linguagem C++.
b) O tipo multiconjunto de um outro tipo X, designado por Mc<X>.
c) O conjunto das variáveis da rede designado por V.
d) Uma função Var(x) que aplica o elemento x no conjunto das suas variáveis de rede. Se X for um conjunto então a função Var(X) corresponde à reunião dos resultados da aplicação da função Var(x) a cada um dos elementos x∈X.
e) Uma função Tipo(f) que aplica uma função no tipo do valor que retorna.
f) Uma função Tipo(v) que aplica uma variável v∈V no seu tipo.
g) Uma função Tipo(c) que aplica uma constante c no seu tipo.
h) Um vínculo de uma variável, designado por um par $\langle v \leftarrow c \rangle \mid v \in V \wedge Tipo(c) = Tipo(v)$.
i) Uma função Lugar: A→L, que aplica arcos em lugares. Para cada arco devolve o lugar a que está ligado.

Relativamente a cada uma destas alíneas importa referir que:

a) F é o conjunto das funções C++ em sentido estrito, ou seja, excluindo as funções do tipo `void f(...)`.

b) A notação, para multiconjuntos, mímica a das classes genéricas[37] da linguagem C++: entre < e > encontra-se o tipo que é parâmetro. Tal deve-se, em parte, ao facto dos multiconjuntos terem sido implementados através de uma classe genérica mas, também, por a notação ser bastante clara.

c) O conjunto de variáveis da rede é constituído pelas variáveis instanciáveis nos arcos de entrada (variáveis de arco) e acções dos lugares, bem como, parâmetros de eventos e acções. Têm obrigatoriamente de pertencer a uma das cores, pertencentes ao conjunto C (vide Quadro 2.2).

e), f) e g) Apenas o "tipo de função" é menos ortodoxo mas muito intuitivo se pensarmos na chamada da função no interior de uma expressão. Considera-se que todas as funções têm um tipo,

[37] Designadas *template* que é palavra reservada.

ao qual pertencem todos os elementos do seu contradomínio. O conjunto de chegada constitui, pois, o tipo da função.

h) Optou-se por utilizar o símbolo ← para separar a variável do seu valor, como forma de tornar mais legível a notação.

Antes de definirmos as CpPNeTS, vamos definir uma nova estrutura da RdP: a *contenda*. Esta estrutura tem a sua origem na classe *conflict* apresentada em [Itmi, s.d.: 673]. Uma contenda é um conjunto de transições que se encontram em conflito estrutural. Uma transição que não se encontre em conflito constitui também uma contenda ou *contenda unitária*. O conjunto das contendas de uma RdP (e em particular de uma CpPNeTS) designa-se por CO e define-se como:

$$CO = \{C \mid (C \in 2^T) \wedge (\forall t1 \in C \ \exists t2 \in (C-t1): {}^\bullet t1 \cap {}^\bullet t2 \neq \emptyset)\}$$

Quadro 2.1 Definição de contenda.

No quadro seguinte apresenta-se a definição formal das CpPNeTS. Após o quadro, encontra-se uma explicação de cada uma das alíneas pelo que se recomenda a leitura paralela dos dois textos.

Uma CpPNeTS é um tuplo CPPNETS = (C, L, T, A, N, Cor, G, E, Va, O, MI, P, EE, Ev, AE, Av, AL, AT) satisfazendo:

(i) C é um conjunto de tipos (classes) denominados cores. Constituem os tipos possíveis para todas as marcas.

(ii) L é um conjunto de lugares.

(iii) T é um conjunto de transições.

(iv) A é um conjunto de arcos, tal que: $L \cap T = L \cap A = T \cap A = \emptyset$.

(v) N é uma função tal que: $N: A \rightarrow L \times T \cup T \times L$.

(vi) Cor é uma função tal que: $Cor: L \rightarrow C$.

(vii) G é uma função que aplica elementos de T em funções ($G: T \rightarrow F$) de modo a verificar:

$\forall t \in T$: Tipo(G(t)) = Booleano.

(viii) E é uma função que aplica elementos de A em funções ($E: A \rightarrow F$) de modo a verificar:

$\forall a \in A$: Tipo(E(a)) = Mc<Cor(Lugar(a))>.

(ix) Seja ALT o conjunto dos arcos com origem num lugar:

$ALT = \{a \mid a \in A \wedge N(a) \in L \times T\}$

Va é uma função que aplica elementos de ALT em subconjuntos de V, verificando:

$\forall a \in ALT$: $Va(a) = \{v \mid v \in V \wedge Tipo(v) = Cor(Lugar(a))\}$.

(x) O é uma função que aplica elementos de T em sequências ordenadas de elementos de A, verificando:

$\forall t \in T$: $O(t) = (a_1, a_2, ..., a_n)$ com $\forall i \leq N : a_i \in ALT \wedge a_1 < a_2 < ... < a_n$.

(xi) MI é uma função que aplica L em multiconjuntos de modo a verificar:

$\forall l \in L$: Tipo(MI(l)) = Mc<Cor(l)>.

(xii) P é uma função que associa a cada transição um valor de prioridade:

$P: T \rightarrow N_0$.

(xiii) EE é um conjunto de eventos externos.

(xiv) Ev é uma função tal que: $Ev: EE \rightarrow 2^V$.

(xv) AE é um conjunto de acções externas.

(xvi) Av é uma função tal que: $Av: AE \rightarrow 2^V$.

(xvii) AL é um conjunto de relações quaternárias tal que:

$AL = \{(l, v, c, a) | l \in L \wedge v \in 2^V \wedge c \in F \wedge Tipo(c) = Booleano \wedge a \in AE\}$

(xviii) AT é um conjunto de relações binárias tal que:

$AT = \{(t, a) | t \in T \wedge a \in AE\}$

Quadro 2.2 Definição de uma CpPNeTS.

(i) C é o conjunto de cores, ou seja o conjunto dos tipos das marcas dos lugares. Podemos agora definir as variáveis de rede (V):

$V = \{v | Tipo(v) \in C\}$

(ii) (iii) (iv) Os lugares, transições e arcos são conjuntos disjuntos e tal como nas RdPCol podem constituir conjuntos vazios. Desta forma é possível verificar se as restantes inscrições da rede contêm erros. Nomeadamente é possível verificar a sintaxe das especificações.

(v) Considera-se uma função que permite obter o nó de entrada e o nó de saída de cada arco. Permite, igualmente, determinar se o arco é de entrada ou de saída relativamente a um lugar ou transição.

(vi) Função que associa uma cor a cada lugar da rede. Cada lugar pode conter marcas de apenas uma cor.

(vii) A função guarda associa uma função que devolve um valor do tipo *Booleano*, a cada transição. Quando a guarda não for especificada, será considerada igual a uma função constante que devolve sempre o valor *verdade*. As variáveis utilizadas na guarda não têm obrigatoriamente de pertencer ao conjunto das variáveis da rede (V). Apesar de tal ser, em principio desejável, visto que a guarda deve ser utilizada como um simples teste à validade dos valores das variáveis vinculadas pelos arcos de entrada da transição, não se impõe aqui essa restrição. Tal fica ao critério do utilizador.

(viii) Associa a cada arco, uma função que devolve um valor do tipo multiconjunto do tipo associado ao lugar ao qual o arco se encontra ligado.

(ix) Cada arco de entrada tem um conjunto (potencialmente vazio) de variáveis de rede que são por ele vinculadas. Estes vínculos são efectuados com base nos valores das marcas presentes no lugar de entrada pelo que as variáveis são obrigatoriamente do tipo associado ao lugar.

(x) Cada transição tem associada uma ordem de avaliação dos seus arcos de entrada. Estes podem

utilizar valores de variáveis de rede vinculadas por arcos que os precedem nesta ordem.

(xi) Cada lugar contem uma marcação inicial. Esta é constituída por um multiconjunto constante, de marcas da cor associada ao lugar.

(xii) É possível atribuir a cada transição um valor de prioridade. A ausência deste é equivalente à especificação de um valor 0. A prioridade entre transições apenas tem significado quando estas pertençam à mesma contenda.

(xiii) (xiv) Cada evento externo corresponde a uma função que tem por parâmetro, um conjunto (potencialmente vazio) de variáveis de rede.

(xv)(xvi) Cada acção externa corresponde a uma função que têm por parâmetro, um conjunto (potencialmente vazio) de variáveis de rede.

(xvii) Cada lugar tem associado um conjunto de acções. Estas são invocações de acções externas condicionadas pela verificação de uma condição, especificada por uma função *Booleana*. Para cada combinação de vínculos possíveis das variáveis da lista associada, a condição é avaliada e, se verdadeira, a função externa é invocada.

(xviii) Cada transição pode ter uma função externa associada. Essa função é invocada sempre que a transição dispare.

Não é obrigatório que aquando da análise ou simulação da rede, as funções externas (eventos e acções) sejam executadas. A sua execução pode ser substituída pela análise de um conjunto de combinações possíveis para os eventos e/ou pela simples indicação da sua actuação. É o que sucede na geração da máquina de estados a partir da rede sincronizada, em que são consideradas todas as combinações de eventos relevantes[38] e as acções são apenas colocadas na especificação da máquina de estados de forma a que sejam chamadas aquando da implementação da máquina de estados.

[38] Os eventos relevantes em cada instante de análise são aqueles que se encontrem associados a pelo menos uma transição apta.

2.1.2 Comportamento das CpPNeTS

O comportamento das CpPNeTS é seguidamente apresentado, segundo uma ordem e uma nomenclatura, sempre que possível, semelhantes às utilizadas em [Jensen, 92] para a exposição do comportamento das RdPCol. Pretendeu-se desta forma evidenciar as semelhanças e as diferenças entre ambas as redes. A definição do comportamento das CpPNeTS aqui apresentada, refere-se a uma rede que não contem marcas temporizadas. Desta forma facilita-se a comparação com as RdPCol. Na secção seguinte veremos como uma simples modificação à definição aqui apresentada é suficiente para nela ficarem incluídas as CpPNeTS com marcas temporizadas.

No quadro seguinte definem-se algumas funções utilitárias:

Ae é uma função que aplica cada nó x no conjunto dos seus arcos de entrada:
$\forall x \in L \cup T: Ae(x)=\{x': \exists a \in A: N(a)=(x', x)\}$.
As é uma função que aplica cada nó x no conjunto dos seus arcos de saída:
$\forall x \in L \cup T: As(x)=\{x': \exists a \in A: N(a)=(x, x')\}$.
A é uma função que aplica cada par de nós (x_1, x_2) no conjunto dos arcos que os interligam:
$\forall (x_1, x_2) \in (L \times T \cup T \times L): A(x_1, x_2)= \{a \in A \mid N(a) = (x_1, x_2)\}$.

Quadro 2.3 Definição das funções Ae, As e A.

As variáveis de um arco *a* (Var(a)) e de uma transição *t* (Var(t)) são definidas por:

$\forall t \in T \; \forall a \in O(t) \subseteq A: \; Var(a) = Va(a)$.
$\forall t \in T: \; Var(t) = \{v \mid v \in Var(Ae(t))\}$.

Quadro 2.4 Variáveis de arco e variáveis de transição.

O conjunto das variáveis de um arco de entrada, da transição *t*, é constituído pelas variáveis vinculáveis pelo próprio arco. O conjunto das variáveis da transição *t*, é constituído pelas variáveis dos respectivos arcos de entrada.

Como se permite a existência de mais do que um arco entre dois nós x_1 e x_2, define-se $E(x_1, x_2)$ como a expressão resultante da soma das expressões calculadas pelas funções dos arcos, também designadas por expressões dos arcos:

$$\forall (x_1, x_2) \in (L \times T \cup T \times L): E(x_1, x_2) = \sum_{a \in A(x_1, x_2)} E(a)$$

Um vínculo de uma transição *t* corresponde a um conjunto de vínculos das suas variáveis. Cada um dos arcos de entrada de *t* é responsável pelo "fornecimento" de um subconjunto dos vínculos de *t*. Como apenas um dos arcos de entrada de uma dada transição pode impor um vínculo a uma

variável dessa transição, podemos referir-nos aos vínculos das variáveis como *vínculos de arco*. Estes são, portanto, subconjuntos do conjunto dos vínculos das variáveis da rede (Quadro 2.5).

Um vínculo de um arco é uma função definida em V, tal que:
$\forall v \in Va(a)$: $vín(v) \in Tipo(v)$.
O conjunto dos vínculos de um arco (Vín(a)), é definido como:
$\forall t \in T \ \forall a \in O(t)$: $Vín(a) = \{<v \leftarrow c> | \ v \in Va(a), Tipo(c) = Tipo(v)\}$

Quadro 2.5 Vínculo de um arco.

Um vínculo de transição é constituído por um vínculo de cada um dos seus arcos de entrada que sejam responsáveis pela vinculação de variáveis. Aquando da discussão da implementação do cálculo dos vínculos das transições será apresentada uma definição baseada nesse facto.

Um vínculo de uma transição *t* é uma função vín definida em O(t), tal que:
(i) $\forall v \in Var(t)$: $vín(v) \in Tipo(v)$.
(ii) G(t)<vín>.

Vín(t) designa o conjunto de todos os vínculos de t.

Quadro 2.6 Vínculo de uma transição.

Nas RdP de baixo-nível a marcação de cada lugar corresponde a um valor *Booleano* (RdP ordinária) ou a um inteiro positivo ou igual a zero (RdP lugar-transição). Nas RdP de alto-nível e designadamente nas RdPCol e nas CpPNeTS, a marcação de um só lugar pode conter muito mais informação para além da simples quantidade de marcas. Daqui resultam as definições de elementos de marca, elementos de vínculo, marcação e passo.

Os *elementos de marca* correspondem a pares (l, c), onde $l \in L$ e $c \in Cor(l)$.
O conjunto de todos os elementos de marca é designado por EM.

Os *elementos de vínculo* correspondem a pares (t, vín), onde $t \in T$, $vín \in Vín(t)$.
O conjunto de todos os *elementos de vínculo* é designado por EV.

Uma marcação é um multiconjunto finito de elementos de EM.
O conjunto de todas as marcações é designado por **M**.

Um *passo* é um multiconjunto não-vazio e finito de elementos de EV.
O conjunto de todos os passos é designado por **Y**.

Quadro 2.7 Elemento de marca, elemento de vínculo, marcação e passo.

Os vínculos de transição vão vincular as expressões dos arcos de entrada e de saída (da transição), especificando assim quais as marcas que serão subtraídas ou adicionadas aos lugares conforme se trate de lugares de entrada ou de saída, respectivamente. No entanto, para que um passo seja considerado apto é necessário que os lugares de entrada das suas transições contenham as marcas

resultantes da vinculação das expressões dos respectivos arcos:

Dada uma marcação M, um passo Y está apto a ocorrer aquando da ocorrência dos eventos associados às transições dos seus elementos de vínculo, se e só se a seguinte propriedade se verifica:

$$\forall l \in L: \quad \sum_{(t, vín) \in Y} E(l,t) < vín > \leq M(l)$$

Se o passo Y está apto em M, então se (t, vín)∈Y, t está apta com o vínculo vín em M.

Quadro 2.8 Aptidão de um passo.

Se um passo está apto, pode disparar. Tal tem por consequência a modificação da marcação dos lugares de entrada e de saída das transições de cada um dos seus elementos de vínculo.

Quando ocorrem os eventos associados às transições dos elementos de vínculo de um vínculo apto numa marcação M_1, essa marcação é modificada para uma nova marcação M_2, definida por:

$$\forall l \in L: M_2(l) = \left(M_1(l) - \sum_{(t, vín) \in Y} E(l,t) < vín > \right) + \sum_{(t, vín) \in Y} E(t, l) < vín >$$

A primeira soma denomina-se marcas *removidas*, e a segunda marcas *adicionadas*. Diz-se que a marcação M_2 é *directamente alcançável* a partir de M_1 pela ocorrência do passo Y, simbolicamente: $M_1[Y > M_2$. Partindo daqui é possível definir uma *sequência finita de ocorrências de comprimento n*, como:

$M_1[Y_1 > M_2[Y_2 > M_3[\ ... \ M_n[Y_n > M_{n+1}$ com, $n \in \mathbb{N}$, e $M_i[Y_i > M_{i+1}$ para todo o $i \in 1..n$.

Ou seja, uma sequência finita de ocorrências é uma sequência de marcações e passos. M_1 é denominada *marcação inicial* e M_{n+1} é denominada *marcação final*.

De forma semelhante é possível definir uma *sequência infinita de ocorrências*:

$M_1[Y_1 > M_2[Y_2 > M_3[\ ...$

O conjunto de todas as sequências finitas de ocorrências é denominado OSF.
O conjunto de todas as sequências infinitas de ocorrências é denominado OSI.
O conjunto de todas as sequências de ocorrências é denominado OS = OSF∪OSI.

Podemos agora generalizar a noção de alcançável:
Uma marcação M'' é *alcançável* a partir de uma marcação M' se e só se existe uma sequência finita de ocorrências que tem M' por marcação inicial e M'' por marcação final.
Para mais detalhes sobre esta notação aconselha-se [Jensen, 90: 78]

Quadro 2.9 Modificação da marcação resultante do disparo de uma transição. Noção de alcançabilidade. Sequências finitas e infinitas de ocorrências.

Tal como sucede nas RdP ordinárias, não existe uma marcação intermédia correspondente a uma situação em que já se removeram as marcas dos lugares de entrada, mas ainda não se adicionaram marcas aos lugares de saída. Um passo é indivisível.

2.1.3 Temporização nas CpPNeTS

Conforme anteriormente descrito (em §1.7.2), nas RdPCol temporizadas considera-se um atributo de cor do tipo inteiro positivo que se associa ao tempo em que a marca irá estar disponível para os arcos de saída do lugar. Nas CpPNeTS utiliza-se essa potencialidade mas o valor do atributo tempo apresenta uma semântica distinta, igual à proposta em [Gomes et al., 92]. O atributo especifica o tempo que falta para a marca se tornar disponível, ou, por outras palavras, a quantidade de tempo durante a qual a marca não se encontra disponível. Em cada instante de análise, todos os atributos *duração*, de todas as marcas temporizadas, que apresentem valores superiores a zero, são subtraídos de uma unidade[39].

Como as CpPNeTS podem ser utilizadas como redes não-temporizadas, optou-se por uma apresentação formal que não contemplasse o tempo. Numa CpPNeTS qualquer marca não-temporizada é equivalente a uma marca temporizada, com todos os atributos iguais ao da marca não-temporizada, e cujo atributo duração seja constante e igual a zero (sempre disponível). Como tal, torna-se possível uma definição das CpPNeTS que abarque a sua capacidade de especificar temporizações, através de uma simples modificação da definição de passo apto[40] (Quadro 2.8). Para tal basta que nessa definição consideremos, não a marcação, mas sim a *marcação disponível*[41].

A marcação disponível de um lugar l, define-se como:
$\forall l \in L: Md(l) = \sum_{s \in M(t) \wedge s.duração = 0} m(s)'s$, em que *s.duração* denota o atributo *duração* de *s*.

Quadro 2.10 Marcação disponível.

As CpPNeTS são redes com capacidade de modelar temporizações. Utilizar, ou não, esse recurso encontra-se inteiramente ao critério do utilizador. Tal resulta directamente, da já referida equivalência entre marcas temporizadas e não temporizadas. É importante notar que esta equivalência não obriga, em termos de implementação, a considerar que todas as marcas contêm um atributo duração que no caso das marcas não-temporizadas não iria ser utilizado, desperdiçando dessa forma espaço em memória. Tal pode ser evitado à custa de um aumento do tempo de processamento, correspondente ao teste da temporização ou não de cada marca. No caso de se

[39] É o que sucede, por exemplo, na determinação da máquina de estados §4.6.

[40] E, consequentemente, de transição apta.

[41] *ready* na literatura em língua inglesa.

utilizar uma linguagem orientada-por-objectos esse teste será, provavelmente, implementado por vinculação dinâmica[42] [Meyer, 88][Booch, 94]. Dessa forma a aplicação não necessita saber quais as marcas que são, ou não, temporizadas. As vantagens daí decorrentes deverão ser suficientes para compensar o tempo de teste necessário.

2.1.4 Sincronização e actuação nas CpPNeTS

As CpPNeTS suportam o modelo de RdP sincronizadas descrito em [David, 91] e [David et al., 92] e no qual se baseia a extensão proposta em [Gomes et al., 92]. Tal é, no entanto, opcional, ou seja, fica ao critério do utilizador utilizar a capacidade de especificação de sincronismo e actuação. De igual modo, os procedimentos da biblioteca podem ser realizados quer pressupondo uma rede que especifique eventos e actuações, quer considerando uma rede autónoma semelhante às descritas em [Jensen, 90].

Os eventos traduzem funções das variáveis de entrada do controlador. As acções traduzem actuações das saídas do controlador, em função do seu estado actual (marcações da CpPNeTS).

As CpPNeTS foram desenvolvidas tendo por principal objectivo expandir as capacidades de modelação das RdPCol de forma a contemplarem a modelação de controladores, conforme descrito em [Gomes et al, 93] e [Gomes et al., 95]. Na Figura 2.2 apresenta-se a estrutura de um controlador desse tipo, onde se assinalam as "localizações" relativas dos eventos e acções da CpPNeTS. Como facilmente se verifica por observação da figura, o modelo suportado pelas CpPNeTS corresponde ao módulo "Processamento". O módulo "Estado actual" tem uma correspondência directa com a marcação da rede.

A Figura 2.2 sugere a possibilidade de implementação do controlador através da execução da RdP. No entanto, e conforme já referido, essa aproximação revela-se, tipicamente, demasiado exigente do ponto de vista computacional para RdP de alto-nível. Por isso e tal como sugerido em [Gomes et al., 92], o sistema CpPNeTS-S não obriga o controlador a suportar uma RdP. Em vez disso é feita uma análise *off-line* da rede, para obtenção da máquina de estados correspondente. Essa máquina de estados pode ser, posteriormente, implementada no controlador.

---

[42] *dynamic binding* na literatura em língua inglesa.

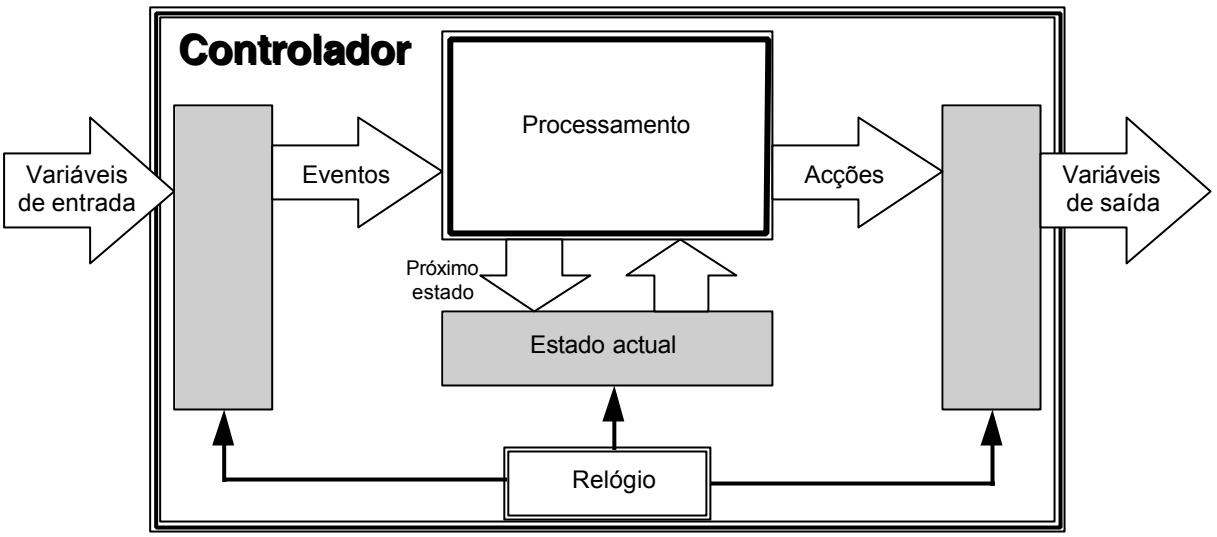


Figura 2.2 Estrutura do controlador assinalando as "localizações" relativas dos eventos e acções. A cinzento assinalam-se blocos de memória.

Eventos

Os eventos são funções externas ao sistema modelado que podem ou não, ser função das variáveis da rede, ou por outras palavras, as variáveis da rede constituem os parâmetros formais das funções externas do tipo *Booleano* correspondentes aos eventos. Desta forma, os vínculos das transições fornecem os parâmetros actuais. Tal possibilita, por exemplo, que o evento corresponda ao teste de um elemento de um vector de variáveis de entrada em que o índice é o parâmetro formal, tal como sugerido em [Gomes et al., 93]: seja *a* um vector de variáveis de entrada, um evento possível é "a[i] == 7".

A parametrização dos eventos externos com variáveis da rede aumenta a capacidade de abstracção do modelo, porque permite representar diferentes eventos (correspondentes a diferentes valores dos parâmetros actuais) como se de um só evento se tratasse. Caso se pretenda a tradução da rede para a máquina de estados correspondente, necessitamos considerar todas as combinações possíveis para os valores dos eventos, para assim prevermos todas as possíveis evoluções. Tal obriga a que os parâmetros associados aos eventos, apresentem obrigatoriamente domínios finitos. Só dessa forma cada evento poderá ser decomposto no conjunto de eventos correspondente a cada uma das combinações dos valores possíveis para os seus parâmetros actuais. Na figura 2.4, apresenta-se um exemplo de uma transição com um evento parametrizado associado.

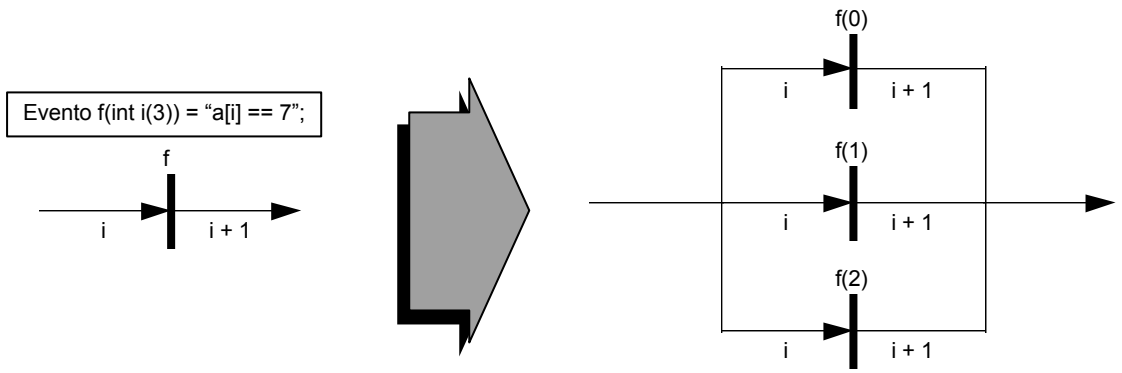


Figura 2.3 Eventos implementados com uma função externa cuja parâmetro formal é uma variável de rede.

A sintaxe utilizada para a definição do evento é semelhante à suportada pela linguagem CpPNeTS-DL (vide §3.2). Como actualmente a linguagem apenas permite parâmetros do tipo inteiro, indica-se a quantidade de valores distintos permitidos para vínculo das variáveis de rede[43]. A transição corresponde, na realidade, a um conjunto de transições em paralelo, cada uma das quais com um dos possíveis conjuntos de vínculos das variáveis parâmetro do evento. No exemplo apresentado, o evento apenas possui um parâmetro com valores que podem variar entre 0 e 2. Como veremos em §4.5.1, não é estritamente necessário multiplicar o número de transições da rede, bastando considerar listas de eventos associadas a cada transição.

Acções

As acções são procedimentos externos ao sistema modelado. As CpPNeTS permitem a especificação de acções associadas às transições e aos lugares. Cada transição pode ter uma ou mais acções associadas. Estas correspondem a procedimentos exteriores que são executados sempre que a transição é disparada. Esses procedimentos podem ter por parâmetros formais variáveis da rede, caso em que o vínculo da transição fornecerá os parâmetros actuais.

As acções associadas aos lugares são executadas condicionadas pela avaliação de uma função *Booleana* denominada condição. As variáveis da lista associada são vinculadas com os valores das marcas presentes no lugar. Todas as combinações de valores são consideradas e para cada uma delas é avaliada a condição e (possivelmente) executada a acção externa. Por exemplo, se o lugar

[43] O objectivo actual é o de permitir a modelação de eventos que sejam função de vector(es) de variáveis de entrada. Os parâmetros podem então corresponder a índices dos elementos do(s) vector(es).

contiver a marcação 2'a + 3'b + 5'c e a lista de variáveis for (i, j), então as combinações de vínculos possíveis serão: <i ← a, j ← a>, <i ← a, j ← b>, <i ← a, j ← c>, <i ← b, j ← a>, <i ← b, j ← b>, <i ← b, j ← c>, <i ← c, j ← a>, <i ← c, j ← b> e <i ← c, j ← c>. Para cada um deste vínculos a condição é avaliada e se verdadeira a acção é executada. A acção pode, por sua vez, ter como parâmetros formais, variáveis de rede, situação em que as combinações de vínculos fornecerão os valores que constituirão os parâmetros actuais.

2.2 A linguagem C++

Para as inscrições nas CpPNeTS e para a construção do pré-processador *pnetcpp*[44] e da biblioteca CpPNeTS-Lib optou-se pela linguagem C++ [Stroustrup, 91] [Stroustrup, 94][Ellis et al., 90][Lippman, 91][Meyers, 92][Meyers, 96][Coplien, 92][Cargill, 93][Cargill, 94a][Cargill, 94b]. Esta opção é justificada por três ordens de factores:

1. É uma linguagem orientada por objectos
2. É uma linguagem muito eficiente.
3. É uma linguagem extremamente divulgada, conhecida e utilizada.

Actualmente, a linguagem C++ suporta todos os passos sugeridos em [Meyer, 88: 60-2] para a "verdadeira orientação por objectos", excepto o terceiro. Este corresponde à gestão automática da memória normalmente designada por recolha-de-lixo automática ou só recolha-de-lixo[45]. As razões para tal ausência são apresentadas em [Stroustrup, 94: 219-22]. Esta omissão obriga a um esforço e atenção adicionais da parte do programador, mas apresenta a vantagem de possibilitar uma maior eficiência de execução.

Uma das principais características de uma linguagem orientada-por-objectos é a noção de classe e objecto. A classe identifica-se com o tipo das variáveis. Estas correspondem aos objectos. Assim sendo, temos "objectos de uma determinada classe" como sinónimo de "variáveis de um determinado tipo". Daqui decorrem algumas das principais motivações para a utilização de uma linguagem deste tipo:

---

[44] petri net to c++ compiler.

[45] Em inglês, respectivamente: *automatic garbage-collection* ou só *garbage-collection*.

- A definição das cores da RdP corresponde à definição de novos tipos de variável, ou seja, de novas classes.
- O conceitos de classe e objecto são fundamentalmente equivalentes aos de páginas e instância de página (nomeadamente macrolugares e macrotransições), nas respectivas superpáginas.

Os seguintes conceitos são característicos de uma linguagem orientada-por-objectos: *modularidade*[46], abstracção de dados, ocultação de informação, herança, polimorfismo e uma forte verificação de tipos. Todos eles foram profusamente utilizados na construção da biblioteca, tornando muito mais fácil a organização e realização das suas estruturas de dados e algoritmos. De resto, as vantagens da programação orientada-por-objectos estão extensamente documentadas sendo amplamente reconhecidas [Meyer, 88][Booch, 94].

Outra característica determinante na opção pela linguagem C++ foi a sua grande eficiência. Conforme afirmado em [Stroustrup, 94: 179], a linguagem C++ é tão eficiente, em termos de tempo de execução (*run-time*) como a linguagem C [Kernigham et al., 88].

O facto de o C++ ser uma linguagem extremamente utilizada e em franco crescimento[47], também pesou muito na opção tomada. De facto, a sua grande divulgação tem vindo a originar o aparecimento de muitas e variadas implementações nos principais sistemas operativos, bem como dos mais variados programas e bibliotecas. Existe também uma extensa bibliografia sobre a linguagem, compreendendo desde manuais de referência, a livros e cursos dos mais diversos níveis, existindo mesmo uma publicação periódica que lhe é exclusivamente dedicada[48]. A *Internet* constitui também uma óptima fonte de informação sobre C++[49].

Actualmente existem já vários trabalhos que relacionam de alguma forma: RdP e a linguagem C++ [Bastide et al., 95][Schöf et al., 95][Pezzè, 94][Raymond et al., s.d.][Manduchi et al., 94]. A tal facto não são certamente estranhas as razões apresentadas.

---

[46] Do inglês *modularity*.

[47] A linguagem C++ é neste momento a linguagem orientada por objectos mais utilizada no Mundo e estima-se que o seu número de utilizadores esteja a duplicar de seis em seis meses [Cline, 96].

[48] A *C++ Report* da SIGS Publications: http://www.sigs.com

[49] Por exemplo, o *site* http://info.desy.de/user/Projects/C++.html é um bom ponto de partida para a exploração do "Mundo do C++" na Internet.

2.3 Alguns trabalhos com pontos em comum

O principal ambiente para a especificação, simulação e análise de RdPCol actualmente existente, dá pelo nome de DESIGN/CPN. É descrito em [Jensen, 92] a par com a definição das RdPCol. Conforme também aí referido, o DESIGN/CPN tem ajudado o desenvolvimento quer da teoria quer das aplicações práticas das RdPCol:

> "Na nossa opinião é extremamente importante que estas três áreas[50] de investigação tenham sido desenvolvidas simultaneamente. As três áreas influenciam-se mutuamente e nenhuma delas poderia ser adequadamente desenvolvia sem as outras duas" [Jensen, 92: V-VI].

O DESIGN/CPN constitui o maior esforço até agora realizado, para a divulgação das RdP e em particular das RdPCol, fora da comunidade científica. Mais de 40 homens-ano foram já dedicados à feitura do sistema e este continua em constante desenvolvimento ao mesmo tempo que as técnicas de análise vão sendo estudadas do ponto de vista teórico. Em [Jensen, 95] apresentam-se os principais resultados desse trabalho. Aguarda-se para breve a publicação de um terceiro livro sobre aplicações práticas do ambiente DESIGN/CPN. Até Janeiro de 1996 o preço do sistema tornava-o proibitivo para muitos dos potenciais compradores[51], mas recentemente o sistema passou a ser distribuído livremente embora sem suporte técnico. As características do ambiente DESIGN/CPN distinguem-no significativamente do sistema CpPNeTS-S devido, fundamentalmente, a três razões:

1. A linguagem utilizada para as notações no DESIGN/CPN é uma linguagem baseada na linguagem *Standard ML* [Milner et al., 90], no CpPNeTS-S é a linguagem C++;
2. As linguagens utilizadas para a implementação do DESIGN/CPN são a linguagem C e a linguagem Standard ML[52]; o CpPNeTS-S utiliza apenas a linguagem C++ para implementação[53].
3. O DESIGN/CPN suporta RdPCol hierárquicas e temporizadas segundo a definição apresentada em [Jensen, 92]; O CpPNeTS-S suporta a especificação de redes sincronizadas.

---

[50] Teoria, implementação de ferramentas (nomeadamente o DESIGN-CPN) e aplicações das RdP-Col.

[51] 24 000 dólares americanos pelo sistema base (12 000 para universidades).

[52] "As partes gráficas das ferramentas CPN estão escritas em C, mas muito do código mais intrínseco está escrito em SML" [Jensen, 92: 172].

[53] Embora o analisador sintático e o parser utilizados gerem código na linguagem C, tal não é significativo, dada a total compatibilidade, do código gerado, com a linguagem C++.

No seu actual estado de desenvolvimento, o sistema CpPNeTS-S não dispõe de um ambiente que possibilite a edição gráfica da rede e integre os aspectos de simulação, análise e geração de código. Tal resultará da natural e futura evolução do sistema.

Um sistema apresentado em [Bastide et al., 95] possui uma arquitectura muito semelhante à do sistema CpPNeTS-S, apesar dos objectivos serem totalmente distintos: trata-se de um ambiente para o desenho de interfaces conduzidas por eventos. A perspectiva base é a de que as modernas interfaces com o utilizador, baseadas em janelas são na realidade casos particulares de sistemas reactivos e as RdP podem ser utilizadas para os desenhar. Embora perseguindo diferentes fins, ambos os sistemas recorrem a uma classe de RdP de alto-nível e à linguagem C++. A descrição da rede é traduzida para código C++, para posterior ligação com código pré-compilado de forma a gerar um programa executável que realiza a tarefa pretendida. Embora utilizando a linguagem C++ para definir os tipos das marcas, o modelo de RdP, apresentado em [Bastide et al., 95], não se baseia nem em RdPCol nem em RdP sincronizadas. Podem ainda apontar-se outras diferenças muito significativas face ao sistema CpPNeTS-S:

- Existem dois tipos de lugares (além dos lugares "normais" existem lugares especiais, os lugares-evento onde são depositados eventos);
- As transições apenas podem ter um lugar-evento no seu conjunto de lugares de entrada;
- As expressões dos arcos resumem-se a uma variável pelo que as variáveis do arcos de entrada surgem como simples parâmetros formais das transições;
- As redes não possuem capacidade de especificação hierárquica;
- A marcação, nos lugares-evento, pode ser modificada por uma entidade externa.

Para um maior detalhe aconselha-se vivamente a consulta de [Bastide et al., 95].

As *THOR Nets* (*Timed Hierarchical Object-Related Nets*) [Schöf et al., 95], são RdP de alto-nível que utilizam a linguagem C++ para descrever os tipos das marcas e que suportam várias formas de temporização (cada transição tem associadas duas funções de temporização), vários tipos de lugares (estruturados como multiconjuntos, filas, pilhas ou filas com prioridade[54]), vários tipos de arcos

[54] Em Inglês: *multisets*, *queues*, *stacks*, ou *priority queues* [Sedgewick, 88].

(normais, de habilitação, de inibição e de consumo[55]), vários tipos de transição e até estruturação dinâmica hierárquica da rede através de invocação de sub-redes. As *THOR Nets* são de facto muito complexas, tão complexas que a análise formal dos seus modelos é quase impossível sendo necessário simular o modelo para dele se poder retirar informação [Schöf et al., 95: 413][56].

O sistema *Cabernet* [Pezzè, 94] é um ambiente de suporte à especificação de sistemas de tempo-real, permitindo verificar essas especificações. O seu objectivo principal consiste na conciliação da necessidade de diferentes vistas, com o poder de uma análise formal. O sistema é suportado por um núcleo interno que integra aspectos de controlo, funcionalidade, informação e temporização. Esse núcleo baseia-se numa classe de RdP de alto-nível e com temporizações, denominada *ER nets*. Estas, juntamente com a sua capacidade de representar temporizações e propriedades funcionais, são descritas em [Ghezzi et al., 91]. Em [Morasca et al., 91] é feita uma apresentação das suas capacidades de modelação de temporizações, juntamente com uma comparação entre as ER nets, as RdP coloridas [Jensen, 87] e as RdP Predicado-Transição [Genrich, 86]. As RdP utilizadas como notação pelo sistema Cabernet, são uma versão com tipos das *ER nets* e denominam-se *Cab nets* [Pezzè, 94]. Esses tipos são especificados utilizando a linguagem C++ e o mesmo sucede com os predicados, acções e funções de temporização. As *Cab nets* também apresentam algumas diferenças semânticas relativamente às *ER nets*. O sistema permite mecanismos de decomposição hierárquica mas a sua ideia mais inovadora é, provavelmente, a do meta-editor. Conforme referido em [Pezzè, 94: 26-30]:

> "Um dos obstáculos principais à utilização de métodos formais para a especificação de sistemas é devido à própria linguagem formal. A maior parte das notações formais são difíceis de ler e escrever e requerem uma considerável aptidão matemática. Se os peritos no domínio não estão familiarizados com a notação formal escolhida, ou a acham complicada, e preferem utilizar uma diferente notação (talvez semiformal) então as hipóteses da nova notação ser utilizada são muito baixas.[57]".

---

[55] Os arcos de habilitação não consomem as marcas servindo apenas para a habilitação das transições. Caso se utilizem arcos de inibição as marcas impedem a habilitação das transições. Os arcos de consumo retiram todas as marcas do lugar se a transição disparar.

[56] Tal necessidade de simulação (por impossibilidade de um estudo analitica), também se aplica às CpPNeTS, e outras RdP que permitam especificações não restritivas de tempos associados, pois para tal basta que a rede permita quaisquer distribuições probabilísticas para as temporizações das transições, lugares ou marcas [Zurawsky et al., 94: 578][Jensen, 95: 165].

[57] Em Inglês no original.

O sistema Cabernet permite a adaptação da notação a utilizar de um modo formal, ou seja, as novas notações definidas pelo utilizador (em rigor um outro tipo de utilizador que não o utilizador final) têm a sua sintaxe e semântica definidas formalmente. Tal é suportado por uma ferramenta denominada *Meta-editor*. Por exemplo, é possível definir-se a possibilidade de utilizar lugares que se comportem como filas de espera [Pezzè, 94: 18-21]. Em [Pezzè, 94: 26-30] é também apresentado um exemplo baseado em *Statecharts* [Harel, 87][Harel, 88][Lucas, 93] como notação para o utilizador. O sistema também possibilita a adição de novas ferramentas, através do *Tool Generator*. Uma implementação que oferece todas estas facilidades encontra-se disponível através de *ftp* anónimo[58]. Muito fica por referir sobre o sistema pelo que se aconselha ao leitor interessado, a consulta do endereço de *ftp* referido onde encontrará várias referências, entre elas [Pezzè, 94].

Em [Itmi, s.d.] é apresentado, de forma resumida, o desenho de um simulador de RdP estendidas. O desenho é orientado por objectos e a linguagem escolhida é o Common Lisp, sendo utilizada uma sua extensão (CLOS) com vista a possibilitar a programação orientada por objectos.

[Wood et al., 90] contem uma descrição de uma simulador gráfico de RdP generalizadas, implementado segundo uma metodologia orientada-por-objectos. São sublinhadas as vantagens da utilização de uma metodologia com tais características.

Também em [Manduchi et al., 94] são sublinhadas as vantagens da utilização de uma metodologia orientada por objectos na implementação de um ambiente de programação baseado em RdP. O protótipo descrito foi implementado utilizando a linguagem C++.

Em [Bloomquist, 90] são realçadas as vantagens da utilização de técnicas orientadas-por-objectos, na realização de simuladores para sistemas a eventos discretos utilizando RdP.

A linguagem LOOPN [Lakos et al., 91][Keen et al., 93] foi expressamente desenvolvida para a especificação e simulação de sistemas utilizando RdP coloridas e temporizadas. É uma linguagem com algumas características das linguagens orientadas por objectos, tais como, classes, herança e polimorfismo. A rede é definida com base em duas hierarquias de classes: a hierarquia de tipos de marcas e a hierarquia de módulos. Em LOOPN uma rede é especificada através de um ou mais módulos. Os módulos são sub-redes que correspondem a *super-transições*. Uma *super-transição*

[58] ftp://ipeset4.elet.polimi.it/dist/Cabernet

liga-se ao módulo que a contem através de lugares que lhe são passados como parâmetros. Esses lugares pertencem ao módulo que contem a *super-transição*[59]. Cada módulo pode definir funções que permitam o acesso às marcações dos seus lugares. Tal permite a leitura das marcações dos lugares sem que o módulo (sub-rede) se encontre ligado a quaisquer outros módulos [Lakos, 92b]. As transições podem ter acções associadas. É através destas acções que é possível conhecer o comportamento da rede quando executada. É possível associar uma restrição a cada lugar. Essa restrição funciona como um filtro para as marcas do lugar. Apenas as que verificam a restrição são visíveis. Todos os tipos de marcas são definidos como subtipos de outros tipos. O tipo mais elementar, do qual todos herdam directa ou indirectamente, denomina-se *null*. Não contem dados mas contem três funções que por herança, todos os tipos de marcas contêm: *first*, *last* e *delay*. Estas funções permitem a selecção de marcas com base no seu tempo de chegada ao lugar. Com base nesta funcionalidade, é possível associar uma temporização às marcas presentes num lugar. Esta temporização pode ser calculada por uma função definida pelo utilizador. Após a especificação da rede (constituída por um conjunto de módulos), esta é traduzida para código na linguagem C, para, após compilação, ser executada.

O documento http://www.cs.utus.edu.au/Research/opn.html contem uma lista dos sistemas de simulação e/ou análise de RdP actualmente disponíveis. Para cada um dos sistemas são assinaladas as principais características, nomeadamente:

1. Nome e versão do sistema, actualmente disponível.
2. Endereços para obtenção do sistema e respectivo preço.
3. Ambientes computacionais sobre os quais o sistema se encontra disponível e linguagens de desenvolvimento utilizadas.
4. Classes de RdP suportados, funcionalidades, documentação.

[59] Esta forma de interface entre sub-redes é idêntico ao denominado *substitution transitions* [Huber et al., 90][Jensen, 92] que também é utilizado no sistema CpPNeTS-S. Em [Lakos, 92a: 23] encontra-se um exemplo que ilustra bem este mecanismo de interface

Exceptuando os protótipos referidos em [Gomes et al., 93] e [Gomes et al., 94], nenhum sistema, quer dos já referidos, quer dos constantes na lista citada, implementam um suporte à geração de código para controladores, baseado numa classe de RdP coloridas e sincronizadas.

Capítulo 3

O Pré-processador

plus valet actum quam scriptum

Um compilador é um programa que lê um programa escrito numa linguagem (linguagem fonte) e o traduz para um programa equivalente noutra linguagem [Aho et al., 86: 1]. Um pré-processador é um programa que produz um programa que se destina a ser lido por um compilador e consequentemente compilado [Aho et al., 86: 16]. Como veremos, o pré-processador *pnetcpp* que aqui será descrito desempenha as funções *1* (processamento de macros) e *4* (extensões à linguagem) apresentadas em [Aho et al., 86: 16].

A Figura 3.1 ilustra, de forma muito simplificada, o funcionamento do pré-processador *pnetcpp*.

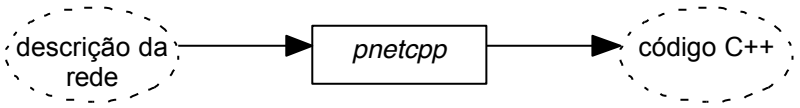


Figura 3.1 O pré-processador *pnetcpp*.

A descrição da rede é efectuada utilizando uma linguagem desenvolvida para o efeito, apresentada em §3.2. O pré-processador gera ficheiros contendo código C++ que para além da descrição da RdP, contêm a invocação de um procedimento contido na biblioteca de suporte *CpPNeTS-Lib* (vide §4).

A título comparativo, no sistema LOOPN, o tradutor é composto por duas aplicações autónomas: um *parser*, responsável pela geração de uma especificação intermédia; e um ou mais geradores de código que traduzem as descrições na respectiva linguagem. Tal é justificado por tornar mais fácil a

criação de outras aplicações auxiliares, nomeadamente um editor gráfico que gere código na representação intermédia e geradores de código para outras linguagens. Essa aproximação poderia ter sido seguida também no sistema CpPNeTS-S mas não o foi por duas razões fundamentais:

1. a tarefa de geração de código é suficientemente simples para poder ser integrada num eventual editor gráfico
2. não se prevê a necessidade de gerar código noutras linguagens que não a linguagem C++, até porque, tal implicaria, na prática, um nova implementação do sistema.

3.1 O pré-processador pnetcpp

O pré-processador *pnetcpp* (*petri net to c plus plus pre-processor*) efectua a tradução da descrição de uma página da rede hierárquica na linguagem *CpPNeTS-DL* (vide §3.2), para código na linguagem C++. Para cada página *p* (normalmente especificada num ficheiro *p.pag*) o pré-processador gera uma classe C++ distribuída por dois ficheiros: um denominado *p.h* e outro denominado *p.c*. Importa salientar que esta divisão do código por dois ficheiros, corresponde a uma convenção extremamente generalizada entre os programadores que utilizam a linguagem C++[60]. A cada página corresponde, portanto, uma classe C++.

O pré-processador suporta redes hierárquicas, constituídas por instâncias de páginas. No entanto, todas as páginas são vistas da mesma forma pelo pré-processador, ou seja, a posição relativa de cada página na hierarquia não é especificada no código *CpPNeTS-DL*. Quando uma página contem instâncias de outras páginas, tal relação traduz-se, no código C++ gerado, por uma classe contendo objectos de outras classes também compiladas com o *pnetcpp*. Desta forma, existe uma correspondência directa entre a hierarquia de páginas, e a relação *has-a* entre classes C++. Além desta relação, o Quadro 3.1 apresenta as principais correspondências entre as várias construções para a especificação de cada página e o respectivo código gerado.

---

[60] Dada a existencia de várias convenções para a nomenclatura dos ficheiros que contem código C++, como por exemplo *.hpp* e *.cpp*, *.h* e *.C*, *.hh* e *.cc*, *.hxx* e *.cxx*, para citar apenas as mais frequentes, optou-se pela utilização das extensões *.h* e *.c* utilizadas na linguagem C. A razão é de ordem prática: é usual os compiladores de C++ disporem de uma opção para compilar ficheiros com a extensão *.c*.

Páginas.	Classes C++
macrotransicões, macrolugares.	objectos de Classes C++ geradas pelo pré-processador
lugares, arcos e transições.	objectos da Biblioteca
lugares porto	apontadores para objectos da Biblioteca
acções, eventos	objectos da Biblioteca

Quadro 3.1 Correspondências entre a especificação e o respectivo código C++ gerado.

De salientar que os lugares porto (de entrada, saída ou entrada-saída) que servem de interface com as outras páginas são representados por referências para os verdadeiros lugares existentes nessas páginas. Não são pois verdadeiros nós mas apenas referências instanciáveis em tempo de execução, quando a rede *pai*, ou seja, a página que inclui outras páginas, é construída.

Opcionalmente, aquando da compilação de uma página *p*, o pré-processador pode gerar um ficheiro (vide Quadro 3.2) contendo:

1. A definição de um objecto da classe `Cppnets` (uma RdP). É essa a página de topo na hierarquia de páginas. Todas as restantes páginas que constituem a rede, devem, directa ou indirectamente, estar incluídas nessa página.
2. Uma função *main* que invoca a função de biblioteca que irá operar sobre o objecto definido realizando a análise pretendida. Esta é indicada através de uma opção adicional *-ax* onde *x* indica a acção a realizar[61].

```
// Code generated by pnetcpp V. 0.9, a Petri Net to C++ translator

#include <stdlib.h>
#include <new.h>
#include "cppnets.h"

#include "p.h"

p net;

int main() {
   net.GenerateStateMachine();
   return 0;
}
```

Quadro 3.2 Ficheiro gerado pelas opções -m -as do pré-processador pnetcpp para uma dada página *p*, para criação de um executável que gere a máquina de estados síncrona, correspondente.

[61] Presentemente a biblioteca disponibiliza a função de geração da máquina de estados síncrona e a função de geração do grafo de ocorrências.

Para qualquer rede constituída por uma ou mais páginas, a opção 1 permite-nos concluir que esse ficheiro de geração opcional é sempre necessário para, e só para, a página de topo da rede hierárquica.

3.2 Linguagem de Descrição - CpPNeTS-DL

Em [Silva, 85] classificam-se as linguagens de definição de RdP em dois grandes grupos:

1. Linguagens algorítmicas, também denominadas procedimentais ou dinâmicas;
2. Linguagens não-algoritmícas, também denominadas não-procedimentais ou estáticas.

As linguagens do segundo grupo consideram-se de especificação. São não-algorítmicas porque não contêm instruções, não podendo por isso determinar um comportamento dinâmico. É neste grupo de linguagens que se encontra a linguagem desenvolvida para a especificação da supracitada classe de RdP.

A linguagem *CpPNeTS-DL* [62], foi desenvolvida com três objectivos em vista:

1. Ser de leitura e compreensão fácil para o utilizador, de forma a que este possa editar descrições de páginas da rede, num editor de texto vulgar.
2. Poder ser facilmente gerada por ferramentas automáticas, nomeadamente por um ambiente de edição gráfico de RdP.
3. Utilizar sempre que possível, as facilidades oferecidas pela linguagem C++.

O primeiro e segundo objectivos correspondem a uma preocupação de facilidade e versatilidade de utilização. Com o terceiro objectivo pretendeu-se não "reinventar a roda", ou seja, não obrigar quer os utilizadores do sistema a aprender uma linguagem muito mais extensa do que a *CpPNeTS-DL*, quer ao aumento de complexidade e dimensão do tradutor.

Devido à utilização da linguagem C++, o sistema dispõe de uma linguagem mais geral e mais testada do que qualquer linguagem *ad-hoc* que para ele fosse expressamente desenvolvida. A sua utilização permitiu manter a linguagem de descrição *CpPNeTS-DL*, simples ao ponto de apenas contemplar a descrição da estrutura da rede. Todas as declarações e definições de variáveis e funções são efectuadas em C++. As próprias marcações iniciais, de cada lugar, e os resultados da avaliação das expressões dos arcos, são especificados utilizando operações sobre multiconjuntos definidas em

[62] Linguagem de Descrição de Redes de Petri. A sua gramática pode ser consultada no apêndice A.

C++. Em resumo: as inscrições da rede são totalmente escritas em C++. Tal é muito facilitado visto a linguagem C++ permitir:

- operadores infixos.
- tipos parametrizáveis (ou classes genéricas).
- funções que operam sobre diferentes tipos de argumentos (polimorfismo).
- várias funções com o mesmo nome (sobreposição[63]).

Resta lembrar, que a linguagem C++ apresenta também as muito significativas vantagens de ordem prática, já referidas no final do segundo capítulo (vide §2.2).

3.3 Código gerado

A estratégia de geração de código numa linguagem de alto-nível que após compilação e ligação, produz, quando executado, o resultado em vista, é uma estratégia bem conhecida e testada. Nomeadamente ferramentas como o analisador lexicográfico *lex* [Levine et al., 92] e o gerador de "parsers" *yacc* [Levine et al., 92] utilizam este tipo de estratégia. A linguagem objecto é nesses casos a linguagem C. Uma estratégia idêntica foi utilizada em [Lucas, 93], utilizando C++, para a implementação de StateCharts.

Tal como as citadas ferramentas, a linguagem CpPNeTS-DL permite a especificação de código directamente na linguagem a gerar (C++). Esse código é especificado nas secções *code* §3.3.2.

Tratando-se a linguagem *CpPNeTS-DL* de uma linguagem não-algorítmica, as suas construções correspondem a definições, descritivas dos vários elementos constituintes da RdP e da forma como se relacionam entres si. Apenas duas das suas construções se não podem incluir nesse conjunto: a declaração *net* e as secções *code*.

3.3.1 A declaração *net*

A declaração `net` é muito simples limitando-se a especificar um nome para a página. Esse nome vai constituir o nome da classe página gerada pelo tradutor. A declaração `net` é obrigatória.

---

[63] Do inglês: *overloading*.

Esta declaração estabelece uma divisão entre a primeira e as segunda e terceira secções `code` (vide §3.3.2).

3.3.2 As secções *code*

A linguagem *CpPNeTS-DL* prevê a especificação de código C++ a incluir nos ficheiros p.h e p.c a gerar pelo pré-processador. É possível indicar três porções distintas de código a incluir em três regiões distintas dos ficheiros:

1. antes da definição da classe (no ficheiro p.h).
2. dentro da definição da classe (no ficheiro p.h).
3. no final do ficheiro p.c (após definição das funções membro[64] da classe).

A Figura 3.2 mostra a localização de cada uma destas porções de código na especificação da página, e no código gerado. Todas as secções são opcionais. Apenas a sua localização relativa à declaração `net` é significativa conforme já referido (vide §3.3.1).

[64] Nome dado às funções contidas numa classe (correspondentes aos *métodos* na linguagem *Smalltalk* [Sethi, 90])

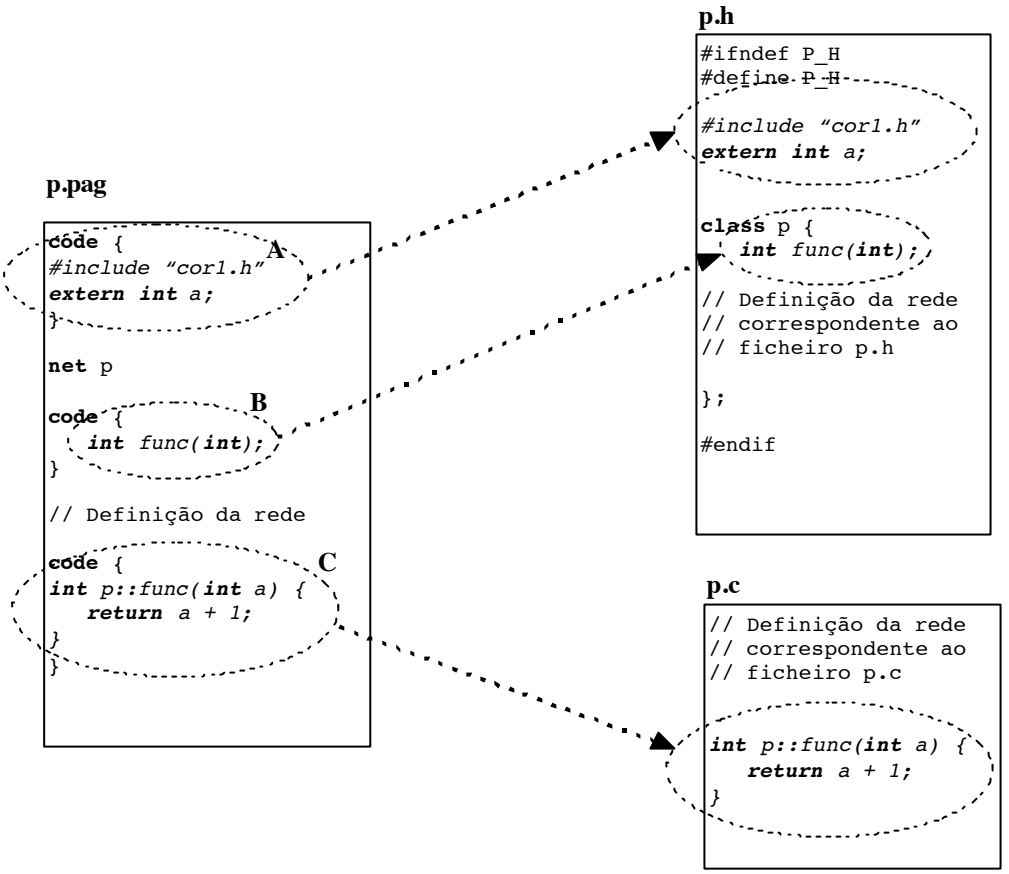


Figura 3.2 Localização nos ficheiros C++ gerados, do código C++ copiado da especificação em *CpPNeTS-DL*. Assinalam-se as três secções que é possível especificar, designadas por A, B e C.

Para além da definição das classes correspondentes às páginas da rede, a linguagem C++ é utilizada para a definição das cores da RdP colorida. A linha "#include "cor1.h", no ficheiro p.pag da Figura 3.2, pretende ilustrar uma possível aplicação dessa primeira secção *code*, precisamente para a inclusão de um ficheiro contendo a definição de uma classe cor. No caso dessa cor ser utilizada apenas pela classe definida no próprio ficheiro, pode ser interessante a sua definição na segunda secção de código. Dessa forma a cor ficará definida no escopo da classe p. Aumenta-se assim, a ocultação de informação[65], um dos princípios fundamentais da programação orientada por objectos.

Por outro lado, é possível utilizar qualquer código C++ juntamente com o código gerado pelo *pnetcpp*. Desta forma é possível uma fácil integração com outras ferramentas futuras que utilizem esta mesma linguagem[66], bem como, a utilização das numerosas bibliotecas C++ existentes para as

[65] Do inglês: *information-hiding*.

[66] Nomeadamente interfaces gráficas com o utilizador, onde a linguagem mais utilizada, e com maior crescimento é, provavelmente, a linguagem C++, aliada por vezes à linguagem C [Kerninghan et al., 88].

mais diversas aplicações. Por outro lado, existe uma quase total compatibilidade da linguagem C++, com a há muito popular linguagem C. Este facto, permite à linguagem C++ aumentar muito significativamente a sua plataforma de apoio. Designadamente aproveitamento do *know-how* dos programadores que conhecendo a linguagem C, podem efectuar uma transição suave para C++, e a utilização de bibliotecas de código C há muito existentes e utilizáveis a partir de código C++.

3.3.3 Definição de variáveis de rede, acções e eventos

As variáveis de rede são definidas utilizando uma sintaxe muito simples e semelhante à oferecida pela linguagem C++. Em C++ as variáveis são definidas fazendo o identificador da variável ser precedido pelo seu tipo (ou classe), por exemplo: `int a` em que `int` é o tipo inteiro e `a` o identificador da variável. Em CpPNeTS-DL a definição de uma variável de rede `v` da cor `C` é especificada da seguinte forma: `variable<C> v` (vide Quadro 3.3 para mais exemplos). A sintaxe corresponde à da definição de uma variável em que o seu tipo é uma classe genérica cujo parâmetro é `C`. A classe C é definida com base na hierarquia de classes que se apresentará mais adiante (vide §4.3).

As acções são definidas utilizando a palavra reservada `action`. Por exemplo, uma acção denominada `act` com dois parâmetros formais do tipo `Int` e `Centro` é definida da seguinte forma: `action act(Int i, Coordenada p.x)`. `i` e `p` são variáveis de rede previamente definidas. `i` é da cor `Int` e `p` é de uma cor que contem um atributo `x` da cor `Coordenada`. Na implementação actual e quando se necessite passar como parâmetro de uma acção ou evento, um atributo de uma dada cor, é necessário defini-lo como variável de rede (vide Quadro 3.3).

Os eventos apresentam uma sintaxe semelhante à das acções, com duas diferenças: utiliza-se a palavra reservada `event` e, na actual implementação, os únicos parâmetros permitidos são os do tipo inteiro. Cada parâmetro especificado tem uma quantidade de instanciações associada. Tal é necessário para que seja possível decompor o evento nos seus eventos elementares (já referido em §2.1.4). Esta quantidade de instanciações é indicada após o identificador do evento (Quadro 3.3).

```
variable<Ponto> p;
variable<Coordenada> p.x;
variable<Int> i;

action act(Int i, Coordenada p.x);

event  ev(int i(3), int j(2));
event  pp(int p.x(4));
```

Quadro 3.3 Definição de variáveis de rede, acções e eventos.

3.3.4 Definição de um lugar

A definição de um lugar é iniciada com a palavra reservada *place*. Segue-se a indicação do tipo das marcas presentes no lugar, um identificador e um campo de texto que pode ser utilizado, por exemplo, para uma descrição da função do lugar na rede.

```
// ...
net sisprod
// ...
code {
   typedef TInt Centro;
}
// ...
variable<Centro> i;
variable<Int> i.v;
// ...
action mover(Centro i.v);
// ...
place <Centro> a "Tapete livre" {
      MS<Centro>(0, 5) + 3 * MS<Centro>(1)
   }
[];
// ...
place <Centro> b "Tapete em movimento" {} [
   (i) {
      return i.v == 2 && i.ttl == 0;
   } mover;
];
// ...
```

Quadro 3.4 Definição de lugares na linguagem CpPNeTS-DL.

A marcação do lugar é efectuada em código C++ e corresponde a uma expressão que quando avaliada deve devolver um multiconjunto de marcas do tipo associado ao lugar. Por último, é possível especificar uma lista de acções associadas ao lugar. Cada uma destas *acções de lugar* é constituída por três partes:

1. Uma lista de variáveis
2. Um bloco de código C++ correspondente ao corpo de uma função *Booleana*.
3. Uma acção da rede.

Nos exemplos apresentados no Quadro 3.4, vemos que o lugar definido com identificador *a*, tem

uma marcação inicial constituída por um multiconjunto com dois elementos: uma marca com o valor *0* e um atributo *duração*[67] igual a *5*, e três marcas com o valor *1* e com o atributo *duração* igual a *0* (valor por omissão). O lugar definido com o identificador *b*, tem uma marcação inicial vazia e uma acção. Esta especifica que: se alguma marca presente no lugar em cada instante de análise for igual a *2* e tiver o atributo *duração* igual a zero (marca disponível), deve ser executada a acção externa `mover` que tem por parâmetro a marca *i*, mais concretamente o seu atributo *v*. Isto porque tratando-se de uma marca temporizada, ela contem também um atributo *duração*.

Dadas as definições do Quadro 3.4, o tradutor gera o construtor C++ do Quadro 3.5.

```
sisprod::sisprod( // ...
                )
{
// ...
AbsPlaceAction** place_a_actions = new AbsPlaceAction*[2];
place_a_actions[0] = new PlaceAction<sisprod>(
   0,
   0,
   this,
   &sisprod::Action0_a_lugarA,
   actions[1]);
place_a_actions[1] = 0;

a = places[posP++] = new Place(
   "a", inst,
   "Tapete livre",
   new MS<Centro>(
      #line 37 "sisprod.pag"
      MS<Centro>(0, 5) + 3 * MS<Centro>(1)),
   place_a_actions);

AbsPlaceAction** place_b_actions = new AbsPlaceAction*[2];
const AbsBindableVariable** var_place_b_action_0 =
   new const AbsBindableVariable* [2];
var_place_b_action_0[0] = new
   TotalBindableVariable(variables[fv + 0]);
var_place_b_action_0[1] = 0;

place_b_actions[0] = new PlaceAction<sisprod>(
   var_place_b_action_0,
   1,
   this,
   &sisprod::Action0_b_moveTapete,
   actions[0]);
place_b_actions[1] = 0;
```

[67] No sistema, o atributo *duração* é denominado *ttl* de "Time To Leave", conforme sugerido em [Gomes et al., 93].

```
b = places[posP++] = new Place(
   "b", inst,
   "Tapete em movimento",
   new MS<Centro>(),
   place_b_actions);

// ...
}

bool sisprod::Action0_b_moveTapete() {
#line 41 "sisprod.pag"
   return i.v == 2 && i.ttl == 0;
}
```

Quadro 3.5 Algum do código do construtor gerado pelo tradutor, para definição dos lugares.

No Quadro 3.5, mostra-se apenas o fundamental para a compreensão da forma como as definições dos lugares são traduzidas para código C++. Para não pormenorizar demasiado esta apresentação, omitiu-se o código gerado no ficheiro com a extensão *.h*, e o código respeitante à definição das variáveis e acções. No capítulo 5 e apêndice E, encontram-se vários exemplos de aplicação.

3.3.5 Definição de uma transição

A definição de uma transição é iniciada com a palavra reservada: *transition*. Segue-se, tal com na definição de um lugar, um identificador e um campo de texto livre que pode ser utilizado para descrever a função da transição na rede. Após o texto, encontra-se a guarda da transição que é especificada por um bloco de código C++ que irá constituir o corpo de uma função da classe-página a gerar.

```
// ...
net sisprod
// ...
code {
   typedef TInt TCentro;
}
// ...
variable<TCentro> ti;
// ...
event ligou();
action mover();
// ...
```

```
transition t5 "Terminar" {
      return ti.t < 1;
   }
   in
      e (ti) {
         return new MultiSet<TCentro>(ti);
      };
      a {
         return new MultiSet<Centro>(ti.t + 1);
      };
   out
      f {
         return new MultiSet<Centro>(ti.t);
      };
   [ligou] [mover] {
      cout << "t5 disparou" << endl;
   }
   priority = 3;
// ...
```

Quadro 3.6 Definição de uma transição na linguagem CpPNeTS-DL.

Os arcos são definidos associados à transição: primeiro os arcos de entrada, cuja definição é introduzida pela palavra reservada *in*, depois os arcos de saída, após a palavra reservada *out*. A definição quer de um arco de entrada quer de um arco de saída, especifica, em primeiro lugar qual o lugar a que o arco se encontra ligado. Seguidamente, e apenas nos arcos de entrada, é possível especificar uma lista de variáveis de rede que serão instanciadas com valores das marcas presentes no lugar, aquando do cálculo dos vínculos dos arcos (vide §4.2.2). A utilização do modificador *bypass_ttl* antes da especificação do nome da variável de arco, possibilita a instanciação desta com todos os valores das marcas presentes no lugar de entrada respectivo, independentemente do valor do seu atributo *duração* (*ttl*). Desta forma, e dado que o valor do atributo duração pode ser utilizado com se de outro atributo qualquer se tratasse (nomeadamente na guarda da transição respectiva), é possível especificar situações de alarme como a apresentada na Figura 1.8. Em §5.1.2 apresenta-se um exemplo de aplicação deste modificador.

No final da definição de um arco, surge o bloco de código que irá constituir o corpo da função C++, responsável pela determinação da expressão do arco. A ordem de especificação dos arcos de entrada define a função *O* (vide §2.1.1). No exemplo do Quadro 3.6, o primeiro arco de entrada apresenta uma variável instanciável (*ti*) que é utilizada pelo arco de entrada seguinte. Assim sendo, a ordem de definição dos arcos não pode ser alterada.

Após a definição dos arcos, é possível especificar um evento e um conjunto de acções externas, associado ao disparo da transição. É também possível, especificar o corpo de uma função a ser executada sempre que a transição dispare. A essa função chamaremos *segmento de código*, dada a

sua identificação com os *code-segments* apresentados em [Jensen, 92: 184-86]. Finalmente é possível especificar uma prioridade associada à transição, utilizando a palavra reservada: *priority*.

O código gerado para a definição de uma transição inclui a definição prévia dos seus arcos. Para a definição dos arcos de entrada pode ser necessária a criação de um vector de referências para as variáveis instanciáveis do arco. As variáveis de rede, os eventos e as acções são definidos de forma autónoma. Desta forma, é possível a sua utilização por várias transições. Os arcos após criados, passam a constituintes do objecto transição a que se encontram ligados (Quadro 3.7).

```
sisprod::sisprod( // ...
                )
{
// ...
// transition t5 input arcs
   AbsInArc** inArcs_t5 = new AbsInArc*[3];
   const AbsBindableVariable** var_arc_e_t5 =
      new const AbsBindableVariable* [2];
   var_arc_e_t5[0] =
      new ParcialBindableVariable(variables[fv + 1], 0);
   var_arc_e_t5[1] = 0;
   inArcs_t5[0] = new InArc<sisprod>(
          this,
          places[fp + 4],
          &sisprod::ArcExp_e_t5,
          var_arc_e_t5, 1);
   inArcs_t5[1] = new InArc<sisprod>(
          this,
          places[fp + 0],
          &sisprod::ArcExp_a_t5);
   inArcs_t5[2] = 0;
// transition t5 output arcs
   AbsOutArc** outArcs_t5 = new AbsOutArc*[2];
   outArcs_t5[0] = new OutArc<sisprod>(
          this,
          places[fp + 5],
          &sisprod::ArcExp_t5_f);
   outArcs_t5[1] = 0;
   Action** actions_t5 = new Action* [1];
   actions_t5[0] = 0;
// transition t5
   transitions[posT++] = new Transition<sisprod>(
          "t5", inst,
          "Terminar",
          inArcs_t5, 2, outArcs_t5,
          this,
          &sisprod::CodeSegment_t5, // code segment
          &sisprod::Guard_t5, // guard
          events[fe + 6], // event
          actions_t5, // actions
          3); // priority
// ...
}

Expression sisprod::ArcExp_e_t5) {
   return new MultiSet<TCentro>(ti);
}
```

```
Expression sisprod::ArcExp_a_t5) {
   return new MultiSet<Centro>(ti.t + 1);
}

Expression sisprod::ArcExp_t5_f) {
   return new MultiSet<Centro>(ti.t);
}

void sisprod::CodeSegment_t5() {
   cout << "t5 disparou" << endl;
}

bool sisprod::Guard_t5() {
   return ti.t < 1;
}
```

Quadro 3.7 Fundamental do código gerado pelo tradutor, para definição das transições e respectivos arcos.

3.3.6 Definição de macrolugares e macrotransições

As macrotransições correspondem às *substitution transitions* referidas em [Huber et al., 90] e [Jensen, 92]. São redes que contêm mais um tipo de nós: os *lugares porto*. Estes são apenas referências instanciáveis para os verdadeiros lugares que se situarão na página[68] que incluirá instâncias da macrotransição. Uma rede que utilize uma macrotransição, vê essa macrotransição como um tipo de nó da rede que embora semelhante a uma transição, apresenta uma forma de conexão diferente da utilizada para as transições. Conforme o já ilustrado na secção 1.7.3, e tal como acontece com os arcos, embora a ligação de uma macrotransição a um lugar possa ser representada, graficamente, por uma seta, na realidade essa seta não corresponde a um verdadeiro arco. Os arcos que interligam de facto a rede com a sua macrotransição, estão definidos no interior da macrotransição. Tal surge de forma muito evidente na definição de uma rede hierárquica na linguagem CpPNeTS-DL. Com efeito, é na definição de uma macrotransição que é estabelecida a sua ligação com a rede. No Quadro 3.8 vemos a definição de uma macrotransição:

[68] Ou em uma sua sub-página.

```
net mesa
// ...
place<int> garfo1 "" {
   MS<int>(1)
} [];
place<int> garfo2 "" {
   MS<int>(1)
} [];
// ...
macro transition filosofo f1(place garfo_direito = garfo1,
                             place garfo_esquerdo = garfo2);
// ...
```

Quadro 3.8 Definição de uma macrotransição em CpPNeTS-DL.

Para além das palavras reservadas `macro` e `transition`, indica-se o tipo da macrotransição (`filosofo`) e um identificador (`f1`). Seguidamente especifica-se a ligação entre a macrotransição e os lugares da rede, ou, mais concretamente, entre os lugares porto da macrotransição e os lugares da rede.

Neste caso a tarefa do tradutor resume-se a incluir um objecto com o nome `f1` da classe `filosofo`, dentro da classe `mesa` e construi-lo aquando da construção de um objecto da classe mesa. Essa construção processa-se em duas fases:

1. invocação do construtor do objecto `f1`
2. afectação das referências `garfo_direito` e `garfo_esquerdo` com os lugares `garfo1` e `garfo2` respectivamente.

A primeira fase é a consequência natural da relação de inclusão entre o objecto da classe `mesa` e o objecto da classe `filósofo`. A segunda fase corresponde à efectiva fusão dos lugares porto da macrotransição com os lugares da rede que a contem, neste caso a `mesa`.

Na definição adoptada nas CpPNeTS, um macrolugar não é mais do que um caso particular de uma macrotransição, mais concretamente, um macrolugar é uma macrotransição sem lugares porto. Embora à primeira vista tal possa parecer estranho, tal resulta da forma como as CpPNeTS são definidas em CpPNETS-DL. Se uma rede que não contem lugares porto é utilizada como sub-rede, a sua ligação com a rede que a contem apenas pode ser efectuada através de arcos entre transições da rede e lugares da sub-rede[69]. Daí resulta que a sub-rede irá apresentar a semântica de um lugar, constituirá, pois, um macrolugar. A definição de uma macrolugar apresenta-se, portanto, mais simples que a de uma macrotransição:

```
net N
// ...
macro place Mac mp;
// ...
```

Quadro 3.9 Definição de um macrolugar em CpPNeTS-DL.

O código gerado para o suporte à utilização de um macrolugar é idêntico ao gerado para uma macrotransição com uma única diferença: a segunda fase não existe.

Nada impede, na implementação actual a utilização de uma macrotransição como se de um macrolugar se tratasse. Desta forma, permite-se a existência de sub-redes que apresentam simultaneamente, características de lugar e de transição, ou seja, são nós aos quais é possível ligar lugares e transições. Tal possibilidade é consequência da implementação efectuada e não constituiu um objectivo em si própria. Apesar da especificação de tais nós poder ser facilmente impossibilitada, nomeadamente através de verificações semânticas da parte do compilador ou da biblioteca, optou-se pela sua manutenção deixando ao utilizador a decisão final sobre o seu interesse prático.

3.4 Algoritmos e estruturas de dados utilizados

O funcionamento do pré-processador pode resumir-se ao apresentado na Figura 3.3.

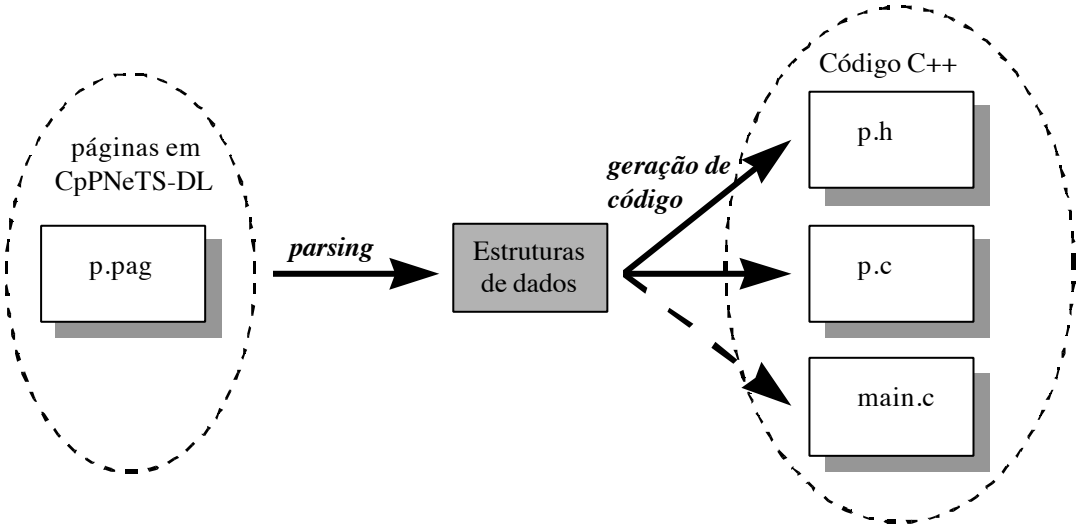


Figura 3.3 Funcionamento do pré-processador *pnetcpp*.

[69] Tal resulta do facto da definição das transições englobar necessariamente a definição de todos os seus arcos adjacentes.

É durante o *parsing* do ficheiro *p.pag* que são preenchidas as estruturas de dados que irão fornecer a informação necessária à posterior geração de código. É também, durante o *parsing* que são verificados os erros de sintaxe e de semântica. A verificação da grande maioria dos erros de sintaxe foi deixada a cargo do código gerado pelo *yacc*, já os erros de semântica foram, necessariamente, tratados caso a caso[70]. Um exemplo de um erro de semântica verificado pelo pré-processador, talvez o mais frequentemente cometido pelo utilizador, consiste na verificação da definição prévia de todos os identificadores utilizados.

As estruturas de dados do pré-processador podem considerar-se divididas em dois grupos: um constituído por uma tabela de símbolos que permite uma busca eficiente dos vários identificadores, aquando do *parsing*, outro constituído por um conjunto de estruturas de dados que reflectem de forma bastante linear, as construções permitidas pela linguagem.

A tabela de símbolos implementada utiliza uma estrutura de dados do tipo tabela de *hash* em que as colisões são colocadas em listas associadas a cada posição da tabela (*hash* aberto)[Cormen et al., 90][Kruse, 84][Sedgewick, 88]. Neste tipo de estrutura o factor determinante de uma maior ou menor eficiência de acessos é imposto pela função de *hash*. Esta deve oferecer a melhor dispersão possível, senso ao mesmo tempo de rápida execução. A função utilizada é a sugerida em [Cargill, 94b].

As restantes estruturas de dados, espelham a estrutura de um ficheiro de especificação. Assim sendo, existem várias estruturas de dados do tipo *lista* correspondentes a:

- Macrolugares
- Macrotransições
- Lugares
- Transições
- Lugares porto
- Acções
- Eventos

Cada uma destas estruturas correspondem a uma lista de objectos do tipo correspondente. Estes objectos contêm, em alguns casos, estruturas semelhantes. Por exemplo, a classe Transição contem,

[70] O yacc não prevê nenhum mecanismo de especificação do tratamento de erros de semântica.

além de outros dados, duas estruturas em lista: uma relativa aos seus arcos de entrada, a outra relativa aos seus arcos de saída.

As estruturas de dados do pré-processador, espelham de uma forma muito nítida a sintaxe da própria linguagem. Este facto, permite uma fácil compreensão e modificação do pré-processador, muito útil para a sua evolução e futuros melhoramentos.

Capítulo 4

A Biblioteca

E mais do que isto
É Jesus Cristo
Que não sabia nada de finanças
Nem consta que tivesse biblioteca...
Fernando Pessoa

A biblioteca é constituída por um conjunto de classes C++ pré-compiladas. Essas classes destinam-se a ser ligadas ao código resultante da compilação das classes geradas pelo tradutor que fornecem a descrição da rede. As classes da biblioteca fornecem o suporte à representação da rede, bem como um conjunto de procedimentos de simulação e análise da mesma. A versão actual da biblioteca oferece dois procedimentos de análise da rede:

1. O gerador da máquina de estados síncrona de uma CpPNeTS com sincronização.
2. O gerador do grafo de ocorrências de uma CpPNeTS autónoma.

Partindo do código C++ gerado pelo pré-processador, obtem-se um programa que quando executado, invoca um dos procedimentos citados, produzindo o resultado pretendido. Os dois procedimentos de análise referidos constituem dois dos algoritmos passíveis de implementação, com base nas primitivas oferecidas pela biblioteca. Outras funções que se podem implementar são, por exemplo: a simulação automática com vários critérios de resolução automática de conflitos; a simulação manual passo a passo em que o utilizador pode intervir em cada passo para alteração do critério de solução dos conflitos ou alteração de algumas das inscrições[71]. Nos exemplos do capítulo

[71] Por exemplo, seria possível a alteração das marcações nos lugares, das guardas, das expressões dos arcos, das prioridades associadas às transições. Este tipo de interacção deve, preferencialmente, ser suportado por um ambiente de edição gráfica da rede e respectivas inscrições que integrará os vários componentes actualmente utilizados: tradutor, biblioteca e compilador de C++.

seguinte, apresenta-se um mecanismo simples de simulação da rede que pode ser facilmente implementado com base nas funcionalidades já disponibilizadas pela biblioteca.

4.1 A rede como resultado de uma hierarquia de páginas

A Rede de Petri é especificada página a página, utilizando a linguagem CpPNeTS-DL. O código gerado pelo tradutor consiste numa classe com o mesmo nome da página que lhe deu origem e descreve a estrutura dessa sub-rede incluindo, nomeadamente:

- A definição dos lugares.
- A definição das transições.
- A definição dos arcos.
- A definição dos lugares porto.
- A definição das sub-redes

E respectivas inscrições:

- Definição das cores.
- Definição das variáveis.
- Expressões dos arcos.
- Associação de cores e marcações iniciais aos lugares.
- Definição de eventos e acções externos.
- Guardas das transições.
- Associação de eventos, acções e guardas às transições.
- Associação de acções aos lugares.

As várias instâncias possíveis de uma mesma página referidas em [Jensen 94] vão, pois, corresponder directamente a objectos.

Os procedimentos da biblioteca que operam sobre a rede não a vêm, necessariamente, como um conjunto de instâncias de páginas que se incluem entre si, apesar de ser esse o modelo de rede que é conhecido do pré-processador[72]. A biblioteca contem, alem da representação hierárquica da rede resultante do código gerado pelo tradutor (classe a classe), uma visão "plana" da RdP, ou seja, uma RdP não-hierárquica. Para que tal seja possível, é necessário converter o modelo conhecido do

[72] Visto ele as compilar independentemente umas das outras.

pré-processador e por ele gerado, constituído por um conjunto de páginas que se incluem mutuamente através da relação *contem*, para o modelo plano, para o qual existe uma RdP não-hierárquica. Isso é conseguido, de uma forma muito natural e eficiente, utilizando os construtores de cada classe gerada.

Construção da hierarquia de páginas

Uma das páginas é escolhida, aquando da sua tradução, para página raíz. Tal é indicado ao tradutor através da opção *-m*. Essa classe irá herdar da classe `Cppnets` (vide §4.2.1); adicionalmente é gerado um ficheiro `main.c` que declara um objecto dessa classe e invoca um ou mais dos seus procedimentos[73].

A representação plana da rede é conseguida através de um conjunto de estruturas de dados contidas na classe `Cppnets`. A página escolhida para raíz da rede hierárquica herda desta classe de forma a conter as estruturas necessárias. Estas estruturas são preenchidas com base nos elementos da rede raíz (lugares, transições, arcos) e nas páginas nela contidas. O construtor da rede raíz, invoca os construtores das sub-redes contidas na sua classe e, em seguida, preenche as estruturas de dados com as referências para os seus próprios elementos (que não são páginas). Cada uma dessas sub-redes actua de igual modo. O código gerado, pelo tradutor, para implementação deste mecanismo pode ser vista nos exemplos do capítulo 5.

Desta forma o próprio mecanismo de construção de objectos em C++ é utilizado para, ao mesmo tempo que os constroi, preencher as estruturas com referências para os elementos de cada página da hierarquia. No final deste processo temos uma hierarquia de objectos tal como especificada pelo utilizador e pelo código gerado pelo pré-processador, mas também uma visão plana da RdP. É esta visão plana da RdP que é utilizada pela biblioteca.

Todas as simulações e análises que se pretendam realizar, podem então ser realizadas sobre uma rede não-hierárquica, com óbvias vantagens do ponto de vista da simplicidade dos algoritmos a implementar[74]. De notar que as estruturas que permitem a vista não-hierárquica da RdP, são simples

---

[73] Em C++, denominados: *funções membro*.

[74] Esta simplicidade poderia, no entanto, ser mantida mesmo sem as estruturas de dados referidas. Bastaria para tal, fornecer um conjunto de funções que permitissem o acesso, em tempo de execução e de forma

referências (apontadores) para os objectos propriamente ditos, pelo que a quantidade de memória que ocupam é pouco significativa.

Interfaces na hierarquia de páginas

A hierarquia de páginas que constitui a RdP, é suportada, no código C++ gerado, por relações do tipo *has-a* entre classes. No entanto, essa relação apenas especifica *como* as classes página se interligam mas não *onde* se interligam. Conforme já anteriormente apresentado (vide §3.3.6), as páginas utilizam lugares porto como pontos de interface entre si. Estes mais não são do que lugares *fantasma* dos verdadeiros lugares com os quais se irão fundir. A sua presença permite a especificação dos arcos de entrada e de saída nas sub-páginas sem que estas contenham os verdadeiros lugares. Esta noção de lugar *fantasma* é perfeitamente traduzida pela noção de apontador para uma posição de memória: a fusão corresponde a afectar o apontador com o endereço do verdadeiro lugar[75]. Desta forma, enquanto que os lugares, pertencentes a cada página, são representados por objectos do tipo *Place*, os lugares porto são representados por apontadores para objectos do tipo *Place*. Aquando da construção dos objectos da classe, esses apontadores não são afectados, visto a classe ainda não conhecer os objectos *Place* com os quais os seus lugares-porto se vão fundir. É na classe que a inclui (super-classe) que esses apontadores são afectados tornando-se referências para os objectos *Place* aí existentes. A ordem de criação de objectos que permite esta interface é então a seguinte (em todas as classes página):

1. Criação dos objectos contidos (sub-classes).
2. Criação dos lugares da classe.
3. Afectação dos apontadores para lugares porto das sub-classes, com os correspondentes lugares criados no passo anterior.

---

transparente a cada um dos elementos presentes nas sub-páginas. O mecanismo seria semelhante ao já apresentado para o preenchimento das estruturas "globais" pelos construtores: primeiro chamar as funções idênticas para os filhos e depois tratar de si próprio. A ineficiência resultante é óbvia: seria necessário percorrer a hierarquia de páginas para encontrar cada um dos elementos o que significaria uma quantidade de acessos à memória tanto maior quanto a profundidade a que esses elementos se encontrassem na hierarquia. Em vez disso, as estrururas globais vêm oferecer um acesso em tempo constante.

[75] A linguagem C++, tal como a sua ascendente, a linguagem C, dispõe de um fortíssimo suporte à noção de apontador.

Um exemplo que ilustra bem a interface entre classes página é o já referido exemplo dos filósofos. No quadro seguinte apresenta-se a especificação da página *mesa* que contem as instancias dos filósofos e correspondente construtor gerado.

```
// a mesa dos filosofos

net mesa

place<Int> garfo1 "" { MS<Int>(1) } [];
place<Int> garfo2 "" { MS<Int>(1) } [];
place<Int> garfo3 "" { MS<Int>(1) } [];

macro transition filosofo f1(place garfo_direito = garfo1,
                             place garfo_esquerdo = garfo2);
macro transition filosofo f2(place garfo_direito = garfo2,
                             place garfo_esquerdo = garfo3);
macro transition filosofo f3(place garfo_direito = garfo3,
                             place garfo_esquerdo = garfo1);

----------------------------------------------------------------
------

// Construtor gerado

mesa::mesa()
:
// macro-transitions
   f1(
          places, transitions,
          variables, pagEvents, actions),
   f2(
          places, transitions,
          variables, pagEvents, actions),
   f3(
          places, transitions,
          variables, pagEvents, actions),
   inst(nInst++)
{
   int fp = places.GetSize();
   int fv = variables.GetSize();
// variables

// actions

// events

// places
   places() = garfo1 = new Place(
          "garfo1", inst,
          "",
          new MultiSet<Int>(
#line 13 "mesa.pag"
 MS<Int>(1) )
          );
   places() = garfo2 = new Place(
          "garfo2", inst,
          "",
          new MultiSet<Int>(
#line 14 "mesa.pag"
 MS<Int>(1) )
          );
   places() = garfo3 = new Place(
          "garfo3", inst,
          "",
          new MultiSet<Int>(
#line 15 "mesa.pag"
 MS<Int>(1) )
          );
// arc expressions
// interfaces
   f1.garfo_direito = garfo1;
```

```
   f1.garfo_esquerdo = garfo2;
   f2.garfo_direito = garfo2;
   f2.garfo_esquerdo = garfo3;
   f3.garfo_direito = garfo3;
   f3.garfo_esquerdo = garfo1;

//Initialization
   Init();
}
```

Quadro 4.1 O exemplo dos filósofos, numa especificação hierárquica de baixo-nível. A ausências de cores é simulada através da utilização de uma cor única (o tipo *int*)com um valor único (*1*). O construtor obriga as páginas contidas a preencherem as estruturas de representação plana da rede.

4.2 Suporte para a rede autónoma

A principal função da biblioteca é a de oferecer um suporte à simulação e análise de uma RdPCol [Jensen, 94]. As características adicionais das CpPNeTS (prioridades associadas às transições e capacidade de sincronismo) encontram-se integradas de forma a só serem utilizadas caso a rede especificada as utilize, ou seja, caso se trate de uma RdP com prioridades ou não-autónoma.

4.2.1 Representação da rede

As classes de suporte à representação da rede, podem considerar-se divididas em dois tipos: classes estruturais e classes auxiliares.

As classes estruturais contribuem directamente para a representação da estrutura da RdP e correspondem quer aos elementos constituintes da RdP (lugares, transições e arcos) quer a formas de estruturar a RdP (contendas).

As classes auxiliares correspondem quer a tipos de dados abstractos de uso geral, como listas e vectores, quer a outros mais específicos que suportam as várias notações da rede. Como exemplo deste último, temos a classe *MultiSet* que permite a manipulação de multiconjuntos como se de um tipo pré-definido se tratasse.

Na figura seguinte apresenta-se um diagrama de classes resumido segundo a metodologia de Grady Booch [Booch, 94]. O diagrama omite os membros de cada classe e algumas classes consideradas pouco importantes para a compreensão do desenho da biblioteca. Desta forma é possível visualizar as relações entre as principais classes do sistema.

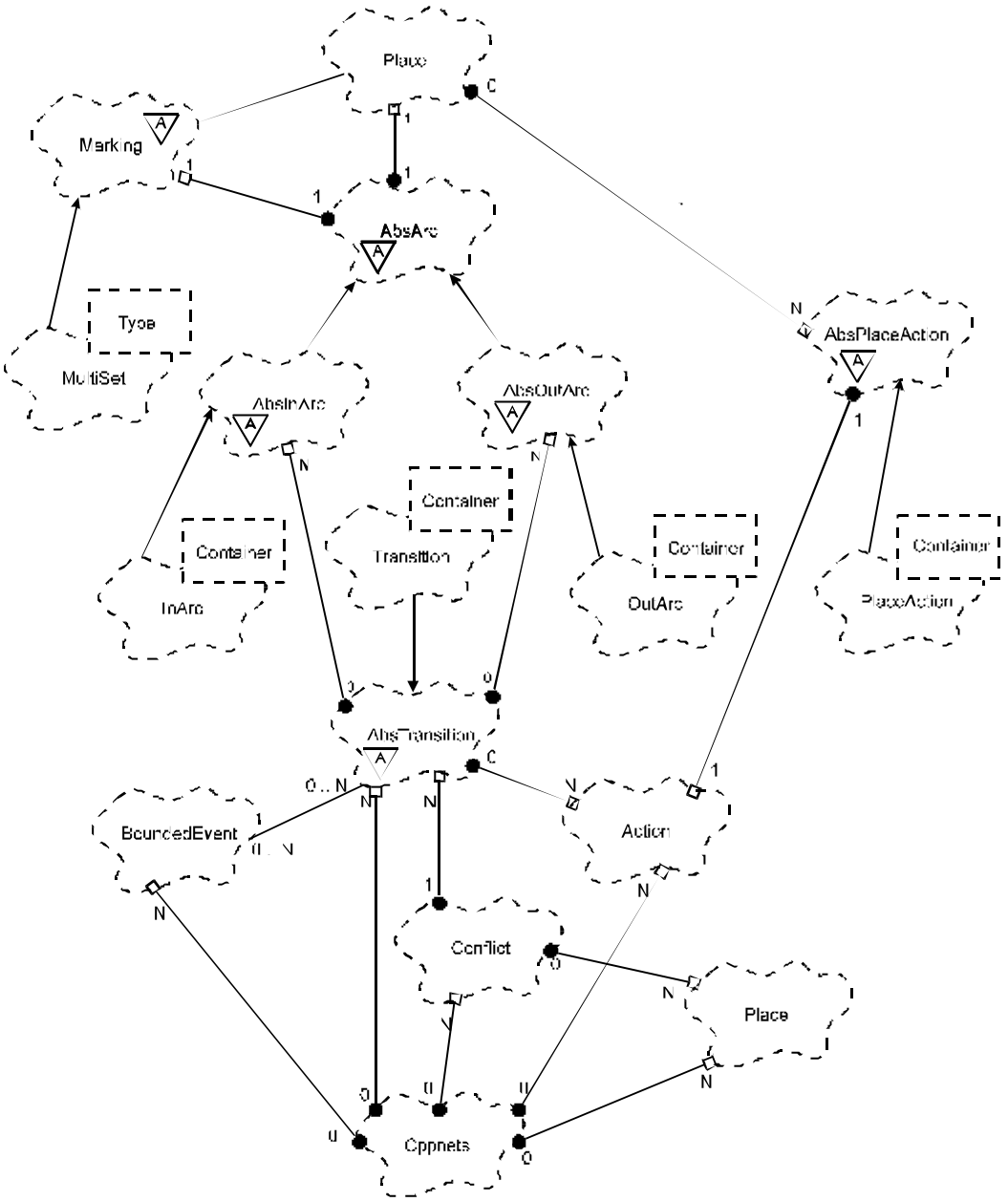


Figura 4.1 Diagrama de classes (resumido) da Biblioteca.

Nota sobre as classes genéricas

É importante assinalar que as classes genéricas são representadas sem que se indiquem as suas classes parâmetro, mais uma vez para simplificar a notação, mas também e principalmente porque as classes parâmetro das classes `InArc`, `OutArc`, `Transition` e `PlaceAction`, são as classes correspondentes às páginas da rede e portanto exteriores à biblioteca. Faltam, portanto: as relações de instanciação entre as classes genéricas e respectivas classes parâmetro; e as relações de "utilização" (*using*) entre as classes parâmetro e as classes genéricas instanciadas.

As classes genéricas `InArc`, `OutArc`, `Transition` e `PlaceAction` constituem pequenas adições às classes `AbsInArc`, `AbsOutArc`, `AbsTransition` e `AbsPlaceAction` respectivamente. Estas últimas contêm a maioria dos dados e do código. As respectivas classes genéricas apenas adicionam apontadores para funções membro de classes geradas pelo tradutor. Como um apontador para função membro exige o conhecimento da classe para poder ser declarado e como neste caso as classes não são conhecidas da biblioteca, optou-se pela adição de classes genéricas que definem esses apontadores. Desta forma é possível diferentes arcos, transições e acções de lugar, possuirem apontadores para funções presentes em diferentes classes, à partida desconhecidas.

Representação das transições

As transições são o centro da estrutura de suporte à rede. Isto porque a evolução da rede é determinada pela aptidão (e prontidão) ou não das transições. As transições contêm duas listas: a dos seus arco de entrada e a dos seus arcos de saída. Cada arco, por sua vez, contem uma referência para o lugar a que se encontra ligado. Logo, os arcos e os lugares são alcançados a partir das transições. Para alem dessas duas listas, as transições contem uma lista de eventos associados e outra de acções associadas. É também em cada transição que são calculados e guardados os seus vínculos. Cada transição tem um identificador igual ao nome do objecto e um campo de texto correspondente à descrição em CpPNeTS-DL.

Representação dos arcos

Os arcos encontram-se divididos em arcos de entrada e arcos de saída dado apresentarem características bastante distintas. Para os arcos de entrada é necessário determinar os vários vínculos possíveis para as suas variáveis. São esses vínculos que irão constituir a floresta de vínculos representativa dos vínculos da transição respectiva. Para os arcos de saída não existe esta necessiadade.

Os arcos de saída apresentam como principal característica a presença de uma variável acumulador. A justificação para tal, resulta do facto das marcações dos lugares de saída de uma qualquer transição, não poderem, em príncipio, ser imediatamente actualizadas aquando do disparo da mesma. Como o disparo das várias transições não é feito em paralelo, mas sequencialmente, nenhuma transição pode modificar a marcação dos seus lugares de saída antes de todas as restantes

terem tido oportunidade de disparar. Como podem existir vários disparos de uma mesma transição em cada passo, esta variável acumula os resultados de sucessivas avaliações das expressões dos arcos. No final, após todas as transições terem efectuado os seus disparos, o valor da variável é adicionado à marcação do lugar de saída.

Representação dos lugares

Cada lugar possui um identificador igual ao nome do próprio objecto na classe (página) onde foi definido. Possui igualmente uma variável do tipo `String` que em príncipio servirá para descrever de forma muito sumária a função do lugar na rede. Cada lugar tem uma lista de acções associada e uma variável do tipo multiconjunto da cor associada ao lugar que constitui a marcação do lugar.

Representação de contendas

Todas as transições que possuam lugares de entrada em comum são agrupadas num mesmo objecto *conflict*. Este corresponde à noção de contenda definida no capítulo 2. Na Figura 4.2 apresenta-se uma RdP e o respectivo agrupamento das suas transições, em quatro objectos *conflict*. Aquando da determinação do passo da RdP, os objectos *conflict* constituem os elementos primordiais. São estes que "disparam". É o seu disparo que impõe o disparo das transições.

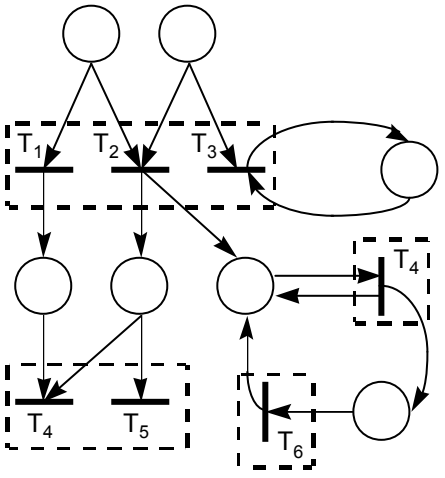


Figura 4.2 Agrupamento das transições em objectos *conflict*. Estes correspondem a conjuntos de transições que partilham directa (e.g. T_1 e T_2) ou indirectamente (e.g. T_1 e T_3) lugares de entrada. Mesmo quando não existe partilha de lugares de entrada considera-se um objecto com uma única transição.

Cada objecto *conflict* contem uma lista de referências para transições. Quando as transições da contenda têm prioridades associadas, essa lista encontra-se ordenada por ordem descrescente.

Dessa forma e aquando do disparo da contenda, basta percorrer a sequência tentando disparar a primeira transição que se encontre apta. Quando não existem prioridades associadas a odem é irrelevante, dado a escolha das transições a disparar ser feita de forma aleatória.

A classe Cppnets

A biblioteca fornece uma classe Cppnets como classe principal no sentido em que representa a rede mais o conjunto de procedimentos que sobre ela foram implementados. Desta forma, e de forma resumida, a classe Cppnets é constituía por:

- Um conjunto de estruturas que permitem a visualização da RdP hierárquica como se de uma rede não-hierárquica se tratasse.
- Um conjunto de procedimentos correspondentes às funções de análise e simulação implementadas sobre a RdP.

A Figura 4.3 apresenta a relação entre o código da class Cppnets e o código gerado pelo tradutor.

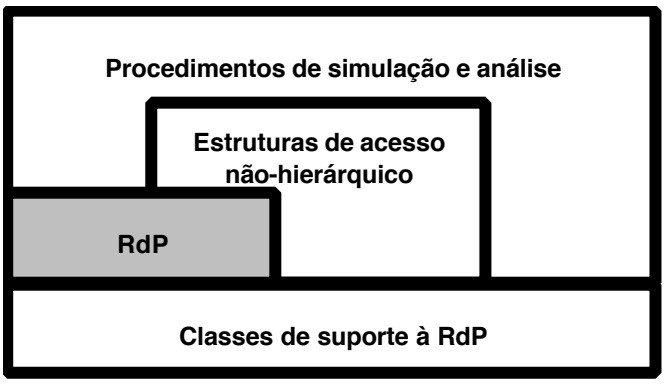


Figura 4.3 Dependencias entre as várias partes constituintes da biblioteca e o código gerado pelo pré-processador. A justaposição horizontal significa utilização. A cinzento representa-se o código exterior à biblioteca que corresponde à especificação da rede, gerada pelo tradutor. O restante código encontra-se contido na classe `Cppnets`.

É importante notar que a especificação da rede proveniente do pré-processador pode também conter procedimentos de simulação e/ou análise que são suportados da mesma forma que os já existentes na biblioteca.

Qualquer rede definida herda da classe Cppnets, passando por isso a conter as respectivas variáveis e funções. Na figura seguinte pode-se visualizar esta relação, bem como, a relação de inclusão (*has-a*) entre as várias classes (seis no exemplo) constituintes de uma rede. Omitiram-se as relações

entre as classes da rede e as classes `AbsTransition`, `BoundedEvent`, `Action` e `Place` por serem iguais às representadas para a classe `Cppnets`.

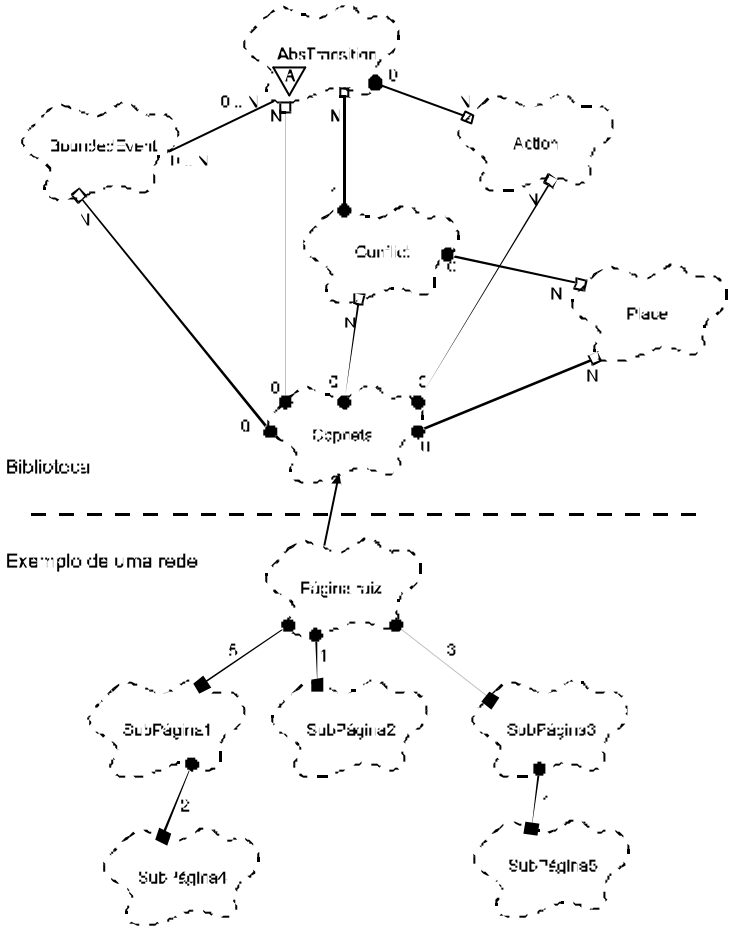


Figura 4.4 Relação entre as classes geradas pelo tradutor e a Biblioteca. Todas as classes da rede apresentam relações com as classes `BoundedEvent`, `AbsTransition` , `Action` e `Place` iguais às representadas para a classe `Cppnets`.

4.2.2 Determinação dos vínculos

A determinação dos vínculos pode considerar-se dividida em três partes:

1. Determinação dos vínculos dos arcos (de entrada)[76] de cada transição. Como vimos, um vínculo de arco é constituído pelos vínculos das suas variáveis. As instanciações das variáveis

[76] Podemos também falar dos vínculos dos arcos de saída de uma transição, mas tal não é significativo visto esses arco se encontrarem (aquando do disparo) sujeitos ao vínculo da transição escolhido para o disparo desta. O vínculo destes arcos não apresenta pois, uma fase de cálculo, posterior ao disparo, por simetria com o cálculo dos vínculos dos arcos de entrada.

vinculáveis, dependem exclusivamente das marcas presentes nos respectivos lugares de entrada.

2. Determinação dos vínculos de cada transição. Estes são subconjuntos do conjunto dos vínculos dos seus arcos de entrada: $V_T \subseteq V_A$. Apenas os subconjuntos de vínculos de arcos que satisfaçam a guarda são considerados.

3. Determinação dos vínculos das contendas da rede. São subconjuntos do conjunto dos vínculos das transições respectivas. Resultam da determinação dos tuplos de vínculos de transição de cada conflito. $V_C \subseteq V_T$.

Seguidamente, analisa-se detalhadamente cada um destes cálculos.

Determinação dos vínculos de um arco de entrada

Um arco de entrada (Lugar→Transição) pode ou não ser responsável pela geração de vínculos. Se a função associada ao arco tem por resultado um multiconjunto constante (sem variáveis livres), então o arco não é responsável pela geração de vínculos. No entanto, parte do grande poder de compactação das RdPCol e por consequência das CpPNeTS advêm, da possibilidade das funções dos arcos poderem apresentar como resultado qualquer uma das marcas presentes no lugar de entrada. Na linguagem CpPNeTS-DL esse efeito é conseguido através da vinculação de variáveis de rede especificadas na definição do arco.

Cada arco contem uma função associada. A partir da avaliação dessa função obtem-se um multiconjunto do mesmo tipo do lugar ligado ao arco. Este multiconjunto pode resultar da avaliação de uma expressão constante, da avaliação de uma expressão contendo uma ou mais variáveis do mesmo tipo dos elementos da expressão, ou ainda resultar de um processamento mais complexo codificado de forma imperativa (na linguagem C++). As variáveis de rede podem, ou não, fazer parte das variáveis utilizadas na função do arco. As variáveis de rede que não sejam vinculáveis pelo arco tê-lo-ão que ser por um dos arcos anteriores. Correspondem pois a variáveis vinculáveis (*valores-esquerdos*) noutro arco préviamente avaliado. Esta ordem de avaliação dos arcos de entrada da transição é necessária de forma a evitar referências circulares entre variáveis dos arcos. Desta forma se dois arcos de entrada utilizam uma mesma variável, apenas um deles (o declarado

em primeiro lugar) é responsável pela sua instanciação[77]. O outro utiliza o valor vinculado pelo primeiro. Este mecanismo encontra-se, naturalmente, generalizado para *n* arcos. É importante notar que esta imposição não diminui o poder de modelação da rede.

Na implementação efectuada, aquando do cálculo de um passo, é calculada uma lista de vínculos para cada arco. Cada vínculo de arco é constituído por um conjunto de vínculos das suas variáveis. Ao vínculo do arco associa-se o multiconjunto resultante da execução da função do arco após aplicação desse conjunto de vínculos e dos conjuntos de vínculos dos arcos que o precedem.

Se se efectuar uma comparação com o simulador/analisador DESIGN-CPN de RdPCol apresentado em [Jensen 92] verifica-se que esse ambiente exige da parte do utilizador um considerável cuidado aquando da especificação das expressões dos arcos das transições. Tal deve-se ao facto de cada variável da transição ter obrigatoriamente de respeitar pelo menos uma das condições enumeradas em [Jensen 92: 193]. Essas condições impedem as especificações apresentadas na Figura 4.5. Em [Jensen 92: 194] estas especificações são apresentadas juntamente com versões modificadas. Estas versões são as únicas que o simulador aí descrito consegue interpretar. A linguagem utilizada para as incrições da rede é uma extensão à linguagem Standard ML, denominada *CPN ML*. Esta é uma linguagem funcional declarativa que oferece o mecanismo de *unificação de padrões*[78] o qual foi largamente utilizado na implementação do algoritmo de cálculo de vínculos, dada a acrescida facilidade de implementação daí decorrente [Jensen 92: 172].

---

[77] Verifica-se: $\forall t \in T \bigcap_{a \in Ae(t)} Va(a) = \varnothing$.

[78] Do inglês: *pattern matching*.

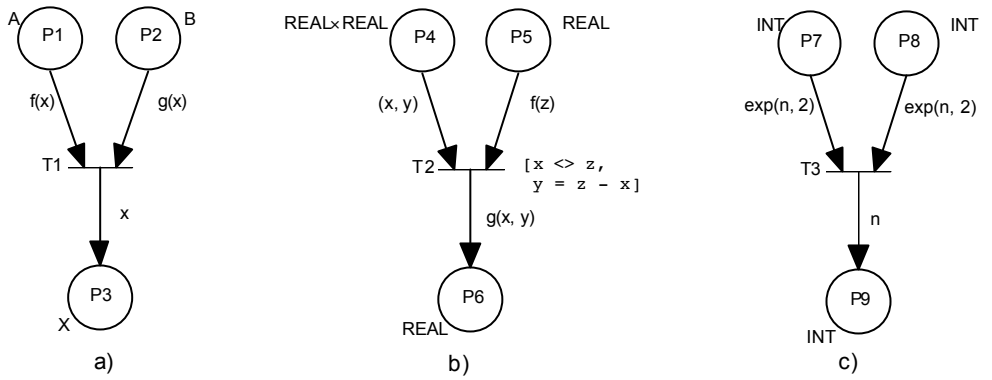


Figura 4.5 Redes com notações que as tornam não interpretáveis pela aplicação DESIGN-CPN [Jensen 92: 194]

Na biblioteca desenvolvida as condições impostas são diferentes. Tal deve-se a uma diferença fundamental entre os algoritmo desenvolvido e o descrito em [Jensen 92]: as variáveis de transição são vinculadas sempre com base directa nas marcas presentes nos lugares de entrada da transição, e nunca a partir de *padrões*[79] (uma construção da linguagem CPN ML [Jensen, 92:192-3]) ou com base no seu domínio. A aplicação DESIGN-CPN não consegue determinar os vínculos da variável x da Figura 4.5a) por ela constituir um parâmetro das funções. Para que a variável seja vinculável é necessário que surja num padrão ou apresente um domínio suficientemente pequeno para que todos os seus valores possam ser testados [Jensen, 92: 193]. Na verdade, a tentativa de vinculação das variáveis com base no seu domínio é utilizada pelo simulador CPN como uma forma de minorar a limitação imposta pela necessidade das variáveis de arco só poderem ser vinculadas a partir de padrões.

Na implementação realizada, no âmbito do presente trabalho, as variáveis nunca necessitam pertencer a domínios finitos e, caso pertençam, tal facto é simplesmente ignorado pelo algoritmo de cálculo de vínculos. As condições que cada variável *v* de transição tem de respeitar, relativamente ao conjunto dos arcos de entrada de uma dada transição, são:

1. *v* é vinculável por um, e só um, dos arcos de entrada.
2. *v* é da mesma cor que as marcas presentes no lugar de entrada do arco que a vincula.
3. *v* não pode ser utilizada como valor-direito numa das funções de arco sem que tenha sido declarada vinculável num dos arcos previamente definidos.

[79] *patterns* no original [Jensen, 92: 193]. Esta mesma referência apresenta a definição de *pattern*.

Estas condições implicam uma notação distinta da apresentada em [Jensen 92]. Por exemplo, para a rede a) da Figura 4.5 duas possíveis notações são apresentadas na Figura 4.6.

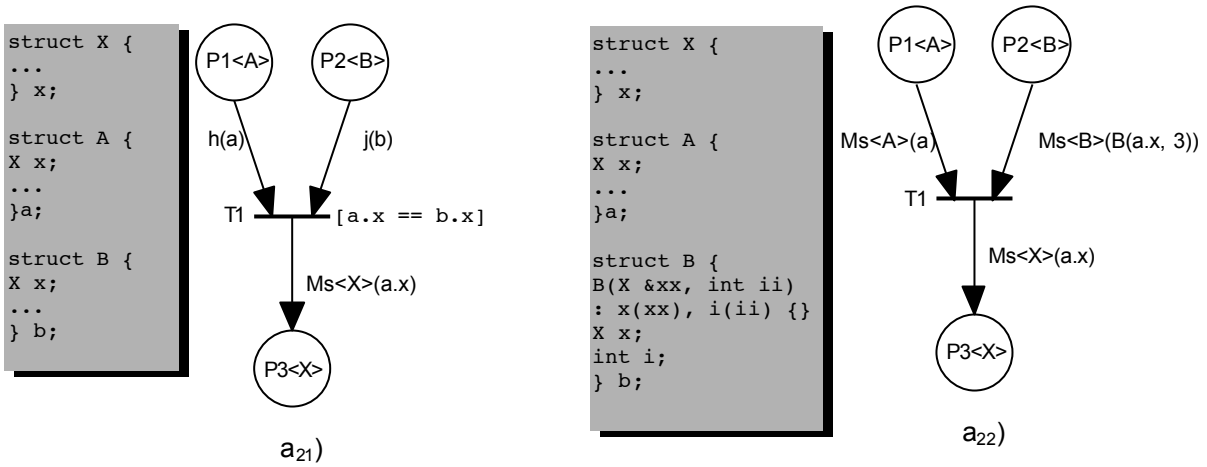


Figura 4.6 Redes equivalentes à da Figura 4.5a) com duas notações possíveis no sistema desenvolvido.

A primeira condição não implica qualquer limitação do poder de modelação. Se se pretende que uma mesma variável seja utilizada em dois arcos diferentes então basta instanciá-la num deles, o outro utilizá-la-á como valor-direito.

Dado que o objectivo da função do arco consiste em "transportar" as marcas presentes no lugar, e dado que a função pode devolver um qualquer multiconjunto de marcas contendo um qualquer número de variáveis, a segunda condição não impõe qualquer limitação ao poder de modelação da rede.

A terceira condição resulta directamente da necessidade da variável conter um valor válido aquando da sua utilização.

Quanto às quatro condições apresentadas em [Jensen 92: 193] e comparado com o algoritmo desenvolvido, constata-se o seguinte:

(i) *v* pode aparecer em qualquer expressão C++ válida. O seu valor não deve, em príncipio, ser modificado por código da responsabilidade do utilizador. Tal só se justificaria em situações muito particulares em que o utilizador pretendesse interferir no mecanismo de disparo da transição. Apenas a biblioteca deve modificar o valor da variável.

(ii) *v* pode pertencer a qualquer classe (cor) definida pelo utilizador, desde que respeitando as condições resultantes da temporização da rede (vide §2.1.3).

(iii) O valor de *v* pode ser lido e modificado nas guardas. No entanto, tal como referido no ponto (i), não se deve recorrer, em príncipio, à possibilidade de modificação do valor.

(iv) *v* nunca deve aparecer apenas nas expressões dos arcos de saída. Tal implica que a variável não foi vinculada por qualquer arco de entrada pelo que o seu valor será indefinido.

No Quadro 4.2 pode ver-se a sintaxe para a especificação das variáveis de cada arco, na linguagem CpPNeTS-DL.

```
…
in
lugar1 (x) {
   if (x > 0) return new MultiSet<E>(E(x));
   else       return new MultiSet<E>();
};
…
```

Quadro 4.2 Especificação de um arco de entrada. A variável *x* é instanciável neste arco, ou seja, é uma variável de arco (vide Quadro 2.4).

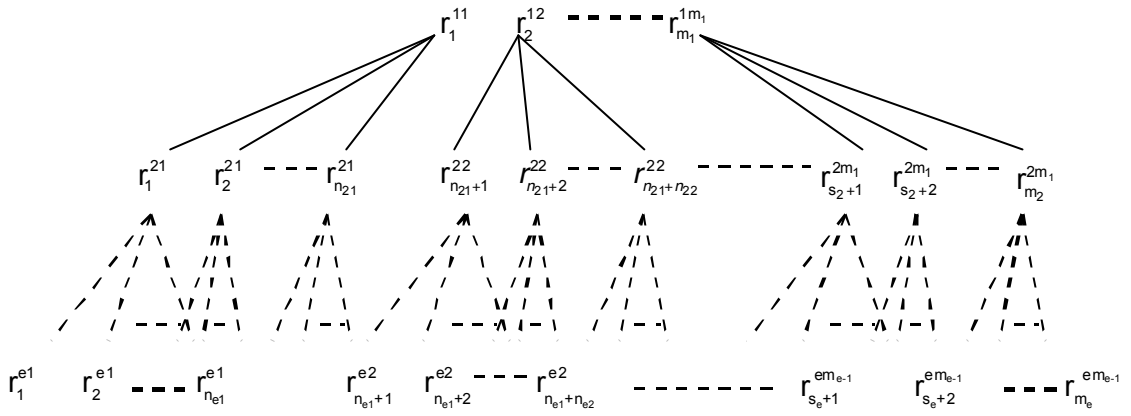
Aquando da avaliação da expressão do arco, esta terá como resultado uma lista de multiconjuntos, resultante das várias instanciações possíveis para os valores-esquerdos especificados. Estes multiconjuntos têm associados a si uma lista de vínculos correspondentes aos valores com que as variáveis de arco foram afectadas. Por exemplo: para um lugar com a marcação [2#'A' + 1#'B'] e um arco cuja função retornasse 1#'A' + x, a avaliação da expressão teria como resultado uma lista contendo dois multiconjuntos: [2#'A']<x ← 'A'> e [1#'A' + 1#'B'] <x ← 'B'>, correspondentes à instanciação de x com 'A' e com 'B' respectivamente.

Determinação dos vínculos de uma transição

Dado que cada arco de entrada de uma transição pode originar várias instanciações possíveis das variáveis da transição, torna-se necessário considerar todos os casos possíveis. Tal implica que para cada resultado da avaliação da expressão do arco se considerem todos os resultados da avaliação da expressão do arco seguinte e assim sucessivamente até ao último arco da sequência. Estamos pois, em presença de um conjunto de estruturas em árvore[80] em que as raizes correspondem aos resultados de avaliação da função do primeiro arco da sequência, o nível seguinte corresponderá

[80] O qual passaremos a designar por "floresta", de acordo a nomenclatura adoptada em [Sedgewick, 88] e [Cormen et al., 90].

aos resultados da função do arco seguinte e assim sucessivamente até ao último arco da transição (vide Figura 4.8).



$$r_i^{ap} = \left(mc_i^{ap}, v_i^{ap}\right) \qquad m_a = \begin{cases} n_{aa} , & a = 1 \\ \sum_{i=1}^{m_{a-1}} n_{ai}, & a > 1 \end{cases} \qquad s_a = m_a - n_{am_{a-1}}$$

com:
i - ordem no nível.
a - número de ordem do arco (nível na árvore).
p - nó pai. Dois nós com igual a e p, têm ambos o mesmo pai.
n_{11} - quantidade de vínculos do primeiro arco na sequência, contradomínio de O (vide Quadro 2.2).
n_{ap} - quantidade de avaliações distintas no nível a para o elemento p do nível a-1.
e - número de arcos de entrada
m_a - quantidade de avaliações distintas do nível a.
mc - multiconjunto resultado.
v - conjunto de vínculos.

Figura 4.7 Vínculos de uma transição. Estrutura de suporte.

Em §2.1.2 foi apresentada uma definição de vínculo de transição. Tomando em consideração a estrutura da Figura 4.7, apresenta-se agora uma definição mais detalhada:

Seja $v_i^a(t) = \begin{cases} v_i^{11} & a = 1 \\ v_i^{ap} + v_p^{a-1}(t) & a > 1 \end{cases}$ e e=#Ae(t), temos que um vínculo v_i de uma transição t é uma sequência de vínculos de arco tal que:

(i) $v_i(t) = v_i^e(t)$
(ii)G(t)<$v_i(t)$>

Um vínculo de transição corresponde a um percurso numa das árvores, que equivale a uma sequência de vínculos de arco que valida a guarda.

```
CálculoDosVínculosDeTransição(&vínculosDeTransição) {
   Pilha pilha;
   int nArco = 0;
   arcosDeEntrada[nArco].CalcularVínculosDoArco();
   se (arcosDeEntrada[nArco].NaoEstáApto()) // o primeiro arco
falhou
      termina; // esta transição não pode disparar
   senão {
      InsereVínculos(vínculosdeTransição, pilha, nArc, RAIZ);
      se (HaMaisArcosDeEntrada(nArco)) { // nArco não é o único
arco
         ElementoDaPilha ep;
         Enquanto(pilha.Tira(ep)) { // pilha não está vazia
            int arcoSeguinte;
            ep.r.Víncula();
            arcoSeguinte = ep.GetNArc() + 1;

arcosDeEntrada[arcoSeguinte].CalcularVínculosDoArco();
            InsereVínculos(vínculosDaTransição, pilha,
arcoSeguinte,
                           ep.Posição());
         }
      }
   }
}

InsereVínculos(&vínculosDaTransição, &pilha, nArco, int pai) {
   Para cada r do arco arcosDeEntrada[nArc] { // r é o vínculo
do arco
      se (arcosDeEntrada[nArco + 1]) { // nArco não é o último
arco
         int posição = vínculosDaTransição.AdicionaNó(r, pai);
         pilha.Põe(ElementoDaPilha(r, posição, nArco));
      }
      senão { // é o último arco
         r.b->Víncula();
         se (GuardaÉVerdadeira()) { // é um vínculo de transição
            vínculosDaTransição.AddicionarFolha(r, pai);
            apta = true; // esta transição está apta
            EscolheEventosSignificativosNesteVínculo();
         }
         senão { // pode-se apagar o vínculo de arco
            apagar(r);
         }
      }
   }
}
```

Quadro 4.3 Algoritmo de determinação dos vínculos de uma transição.

O algoritmo de cálculo dos vínculos de uma transição corresponde à construção da floresta de vínculos de arcos apresentada na Figura 4.7. Esta é construída em profundidade, utilizando uma pilha como estrutura de dados auxiliar. A construção em profundidade permite ir efectuando o vínculo das variáveis à medida que se determina o vínculo de cada arco. Desta forma, quando se alcança uma folha (correspondente a um vínculo do último arco na ordem de avaliação) basta efectuar a vinculação correspondente a esse nó para assim completar o vínculo das variáveis da transição. A guarda pode, então, ser executada e, caso esta seja verdadeira, adiciona-se uma referência para a folha, a uma lista que irá constituir o conjunto de vínculos de transição, ou sejam,

os percursos válidos segundo a guarda. Seguidamente, elegem-se os eventos significativos. Estes são aqueles para os quais o vínculo de transição é válido. Desta forma, é possível rejeitar os eventos com vínculos que nunca se verificam, bem como, os eventos associados a transições não aptas. Tal pode ser extremamente importante para a determinação da árvore de estados, pois aí são consideradas todas as combinações possíveis de eventos.

Determinação dos vínculos de uma contenda

Os vínculos de uma contenda são os vínculos das suas transições disparadas em sequência. Estas correspondem às ordenações possíveis do conjunto das suas transições. Pode existir apenas uma ordenação, a que corresponde uma dessas sequências, no caso em que a contenda contem uma só transição, ou quando tiver sido especificada uma ordem de disparo, através de prioridades associadas às transições.

Pode-se estabelecer um paralelismo entre vínculos de arcos e vínculos de transição, e vínculos de transição e vínculos de conflito. De facto, o vínculo de uma transição é constituído por uma sequência dos vínculos dos arcos dessa transição, e o vínculo de um conflito é constituído por uma sequência de vínculos das suas transições.

Apesar deste paralelismo, o algoritmo para a determinação dos vínculos de um conflito, não apresenta o mesmo tipo de semelhança relativamente ao algoritmo de determinação dos vínculos de uma transição. A razão fundamental resume-se ao facto de cada vínculo de transição corresponder a um possível disparo da transição que por sua vez corresponde a uma possível evolução da rede, enquanto que o disparo de um arco não pode ser considerado isoladamente do disparo dos restantes arcos de entrada da transição. Assim sendo, optou-se por calcular todos os possíveis vínculos de cada transição. Estes correspondem às sequências possíveis de vínculos de arcos. Os vínculos de contenda correspondem a sequências de vínculos de transições. Como os vínculos de transição não obedecem a nenhuma sequência obrigatória de disparo, o cálculo de todos os vínculos possíveis para a contenda seria muito pesado computacionalmente. Pesado ao ponto de não compensar a utilização que depois seria feita, que corresponderia à simples escolha de um possível ramo da árvore de vínculos de transição que seria gerada. Assim sendo, a opção tomada foi a de não calcular vínculos de conflito *a priori*. Os vínculos de contenda são determinados apenas em "tempo de disparo", escolhendo dinâmicamente de entre os possíveis vínculos de transição. Essa

escolha pode já estar constrangida devido à existência de prioridades associadas às transições da contenda, ou ser resultado de algum tipo de estratégia de resolução de conflitos. O facto de todos os vínculos de transição se encontrarem à partida, determinados, juntamente com o facto da eleição dos vínculos a disparar ser feita dinâmicamente, torna extremamente flexível a biblioteca, pois permite a implementação de toda e qualquer estratégia de disparo da rede, sem modificação do código já existente.

4.2.3 Evolução para o passo seguinte — o disparo da rede

O disparo da rede, ou seja, a execução de um passo, pode considerar-se dividido em três níveis de abstracção, cada um deles recorrendo ao nível abaixo:

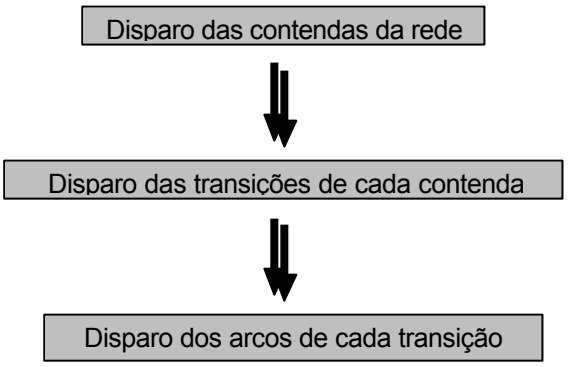


Figura 4.8 O disparo da rede processa-se em três níveis..

Seguidamente, apresentam-se os algoritmos implementados para cada um destes três níveis de disparo. Importa salientar que todas as estratégias de disparo foram realizadas tendo por base o mesmo conjunto de vínculos de transição. Outras estratégias podem ser implementadas sem necessidade de modificar a determinação desses vínculos.

Disparo de uma contenda

O disparo de uma contenda corresponde a uma estratégia de resolução do conflito estrutural que lhe está associado. Na implementação efectuada, implementaram-se três estratégias distintas: uma para a simulação de uma CpPNeTS não-sincronizada; uma para a construção do grafo de ocorrências do mesmo tipo de rede; e uma para a construção da máquina de estados de uma rede sincronizada.

No primeiro caso, efectua-se um disparo de uma ou mais transições aptas. Se existirem prioridades especificadas, os disparos reger-se-ão pela ordenação imposta, caso contrário a escolha da transição é feita aleatóriamente.

Na determinação do grafo de ocorrências, interessa disparar cada um dos vínculos de transição isoladamente dos restantes. Estes disparos unitários irão corresponder aos arcos do grafo. Neste caso o disparo da contenda reduz-se ao disparo de uma das suas transições utilizando um dos seus vínculos.

Na construção da máquina de estados, as transições devem disparar até se encontrarem não aptas. Ou seja, cada disparo da rede (evolução para um estado seguinte) implica o disparo de todas as transições o máximo número de vezes possível. Esta estratégia resulta directamente da definida em [David, 91] para as redes sincronizadas de baixo nível. Aí uma rede (de baixo-nível) sincronizada, evolui para o estado seguinte após o disparo de todas as transições aptas quando o evento ocorre. Como numa rede colorida (e também numa CpPNeTS) cada transição pode substituir um conjunto de transições de baixo-nível, a estratégia equivalente corresponde a disparar todas as transições o máximo número de vezes possível, conforme já apresentado em [Gomes et al., 93]. Como as CpPNeTS permitem a especificação de prioridades associadas às transições, foram implementados dois algoritmos de disparo de transições, um para quando as transições da contenda apresentam prioridades associadas, e outro para o caso contrário (Quadro 4.4).

```
se HáPrioridadesEspecificadas {
   para cada transição t da sequência ordenada de transições {
      se t se encontra apta e é significativa {
         DisparaTotalmenteTransição();
      }
   }
}
senão {
   CalculaConjuntoDeTransiçõesSignificativas(ts);
   Enquanto ts não está vazio {
      SeleccionaUmaTransição(t, ts); // t∈ts  e  ts=ts-t
      se t se encontra apta {
         DisparaTotalmenteTransição();
      }
   }
}
```

Quadro 4.4 Disparo de uma contenda para construção da máquina de estados de uma CpPNeTS sincronizada.

A selecção da transição a disparar é feita aleatoriamente desde que não existam prioridades associadas.

As transições significativas são aquelas às quais se encontra associado, pelo menos, um evento significativo. Este é um evento compatível com algum dos vínculos da transição.

Disparo de uma transição

Implementaram-se duas estratégias distintas para o disparo das transições: uma que escolhe aleatoriamente os vínculos a utilizar e outra que dispara a transição utilizando um dado vínculo. Esta última estratégia é utilizada aquando da construção do grafo de ocorrências, dado que cada arco corresponde ao disparo único de uma transição com um dado vínculo.

Escolhido o vínculo, a que corresponde uma folha da floresta de vínculos de arco, o disparo da transição é efectuado através do disparo dos seus arcos de entrada. Como cada nó contem os multiconjuntos resultantes da avaliação da função do arco respectivo, o disparo da transição pode ser efectuado através de um percurso na floresta, de uma folha para a raíz respectiva. Desta forma é possível simplificar a estrutura de dados de suporte. Em cada nó (vínculo de arco) é efectuado o disparo do arco.

Disparo de um arco

Dado o vínculo de arco, o disparo consiste simplesmente na subtracção do multiconjunto resultado (mc), da marcação do lugar de entrada respectivo. O multiconjunto resultado resulta da avaliação da expressão do arco (definida por uma função C++). A expressão é avaliada após aplicação dos vínculos impostos pelos arcos anteriores na ordem de definição.

4.3 Suporte para a rede temporizada

Uma das principais motivações para a utilização da linguagem C++, foi a possibilidade do utlizador poder definir as cores como classes C++. A adição de temporizações às CpPNeTSs, sob a forma de um atributo *duração*, veio criar a necessidade da biblioteca testar esse atributo com vista a determinar a prontidão das marcas temporizadas. Como o atributo a testar se encontra dentro da definição da cor e essa definição se encontra a cargo do utilizador, a biblioteca não teria, à partida, forma de conhecer o atributo a testar. Além disso, é de todo o interesse, a biblioteca testar a aptidão das transições (que implica verificar a prontidão e aptidão de marcas) de uma forma

uniforme sem se preocupar com o facto das marcas serem ou não temporizadas. A resposta a estas duas questões, consistiu em colocar no interior da Biblioteca a forma de definir e testar as marcas temporizadas de forma transparente quer para o utilizador quer para outras classes da biblioteca. Tal é suportado pela hierarquia de classes que se representa na Figura 4.8, utilizando um diagrama de classes na notação de Booch [Booch, 94].

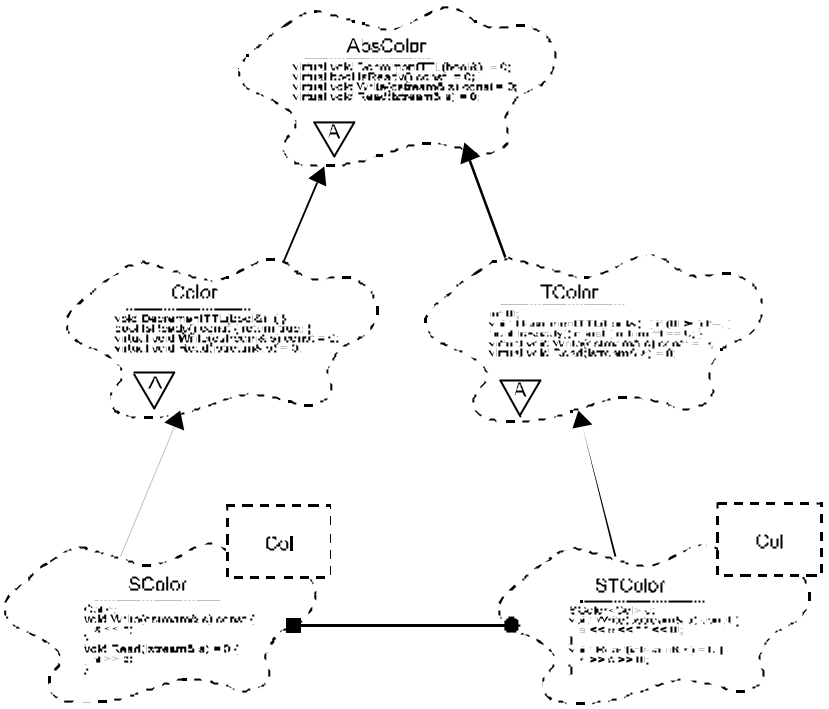


Figura 4.9 Classes de suporte à definição das cores rede. Oferecem temporizações e persistência às classes definidas pelo utilizador.

A classe `AbsColor` é uma classe abstracta que obriga à definição das funções `DecrementTTL`, `IsReady`, `Write` e `Read`. Da classe `AbsColor` herdam duas classes: `Color` e `TColor`. A primeira corresponde a uma cor não-temporizada e a segunda a uma cor temporizada. Apenas a segunda define o atributo `ttl` (*duração*). Como tal definem de forma distinta as funções `DecrementTTL` e `IsReady`. Na classe `Color` a função `DecrementTTL` é nula e a função `IsReady` devolve sempre o valor **`true`**. Já na classe `Tcolor` a primeira decrementa o valor do atributo `ttl` de uma unidade e a segunda devolve **`true`** apenas se o valor do atributo é igual a zero.

As classes `Color` e `TColor` permitem a utilização tranparente de cores temporizadas ou não. Ambos os tipos de marca são testado com a função `IsReady` e os seus `ttls` decrementados com a função `DecrementTTL`. As restantes classes são classes genéricas que facilitam a definição de cores correspondentes aos tipos pré-definidos na linguagem C++. Permitem, nomeadamente, definir as cores das variáveis da rede com simples classes C++, sem incluirem o atributo `ttl`. Caso a temporização seja necessária utiliza-se a classe `STColor`. Por exemplo, se as classes definidas pelo utilizador se denominarem C e D, é possível definir quatro cores distintas: duas temporizadas e duas não-temporizadas (Quadro 4.5). O quadro apresenta também a forma de definir cores directamente

a partir das classes `Color` ou `Tcolor`, utilizando herança.

```
code {
class C {
...
};
class D {
...
};
typedef SColor<C>  CS;
typedef STColor<C> CT;
typedef SColor<D>  DS;
typedef STColor<D> DT;

class E : public Color {
...
};
class F : public Tcolor {
...
};
}
variable<CS> cs;
variable<CT> ct;
variable<DS> ds;
variable<DT> dt;
variable<E> e;
variable<F> f;
```

Quadro 4.5 Definição de cores utilizando as classes genéricas `SColor` e `STColor`.

As classes definidas pelo utilizador têm obrigatoriamente de definir os operadores de entrada e de saída, << e >> respectivamente. Tal é necessário para que os objectos possam ser escritos e lidos de memória secundária de forma transparente para os utilizadores da classe. Dessa forma adicionou-se persistência às classes.

4.4 Uma estrutura de dados com base em objectos persistentes

Tanto a máquina de estados síncrona como o grafo de ocorrências são estruturas que facilmente atingem dimensões consideráveis. Como forma de minorar as limitações impostas pela capacidade de memória RAM disponível, optou-se pela implementação de uma estrutura de dados em memória secundária que serve de suporte à construção da máquina de estados síncrona e grafo de ocorrências. Como é necessário guardar em memória secundária as marcações da rede em cada estado ou nó e lê-las mais tarde, a biblioteca necessita saber escrever e ler, posteriormente e para um diferente endereço de memória, todo e qualquer objecto de qualquer classe definida pelo utilizador e, portanto, desconhecida da biblioteca. Por outras palavras, é necessário adicionar persistência [Booch, 94] [Tichy et al., 94] às cores.

A cada classe multiconjunto de uma cor, encontra-se associado um ficheiro onde são escritos e lidos os seus objectos. São estes ficheiros que suportam a persistência dos multiconjuntos e, consequentemente, das marcações.

A estrutura de dados é constituída por quatro ficheiros mais os ficheiros associados a cada uma das classes de multiconjuntos:

- `states.db` - Contem ternos constituídos por `(apontador para marcação, apontador para acções, apontador para as transições para estado seguinte)`
- `trans.db` - Contem tuplos: `(identificador do estado, combinações de valores de eventos, invocação de acções, lista de apontadores para estados seguintes).`
- `markings.db` - Contêm pares: `(identificador do estado, lista de apontadores para as marcações dos lugares).` As marcações dos lugares são multiconjuntos que são escritos nos respectivos ficheiros.
- `actions.db` - Contem pares: `(identificador do estado, lista de acções a executar).` As acções correspondem a chamadas de função com a estrutura: `NomeFunção(parâmetro1, ..., parâmetroN).`
- `marks?.db` - Contêm marcações dos lugares. *?* é um valor inteiro gerado automaticamente aquando da criação do primeiro multiconjunto de cada tipo. Cada classe de multiconjuntos de um dado tipo, tem um ficheiro deste tipo, associado.

Os ficheiros acima referidos, são utilizados aquando da geração da máquina de estados e do grafo de ocorrências. Outros algoritmos que necessitem de armazenar marcações intermédias em disco poderão também utilizar esta estrutura. Tal já sucedeu com o algoritmo de construção do grafo de ocorrências, dado que a estrutura foi inicialmente pensada como suporte para a realização da máquina de estados.

Os ficheiros encontram-se em formato de texto. Por esta razão, os mesmo ficheiros podem ser facilmente processados por qualquer aplicação que interprete a máquina de estados, nomeadamente utilizando analizadores lexicográficos como o *Lex*. Por outro lado, vem facilitar a depuração da

biblioteca e da especificação da rede.

Paralelamente aos ficheiros da estrutura de dados, é gerado um conjunto de outros ficheiros num formato de fácil leitura que permite quer a verificação manual, quer o processamento automático dos resultados obtidos. Estes ficheiros são descritos após a apresentação do algoritmo de construção da máquina de estados.

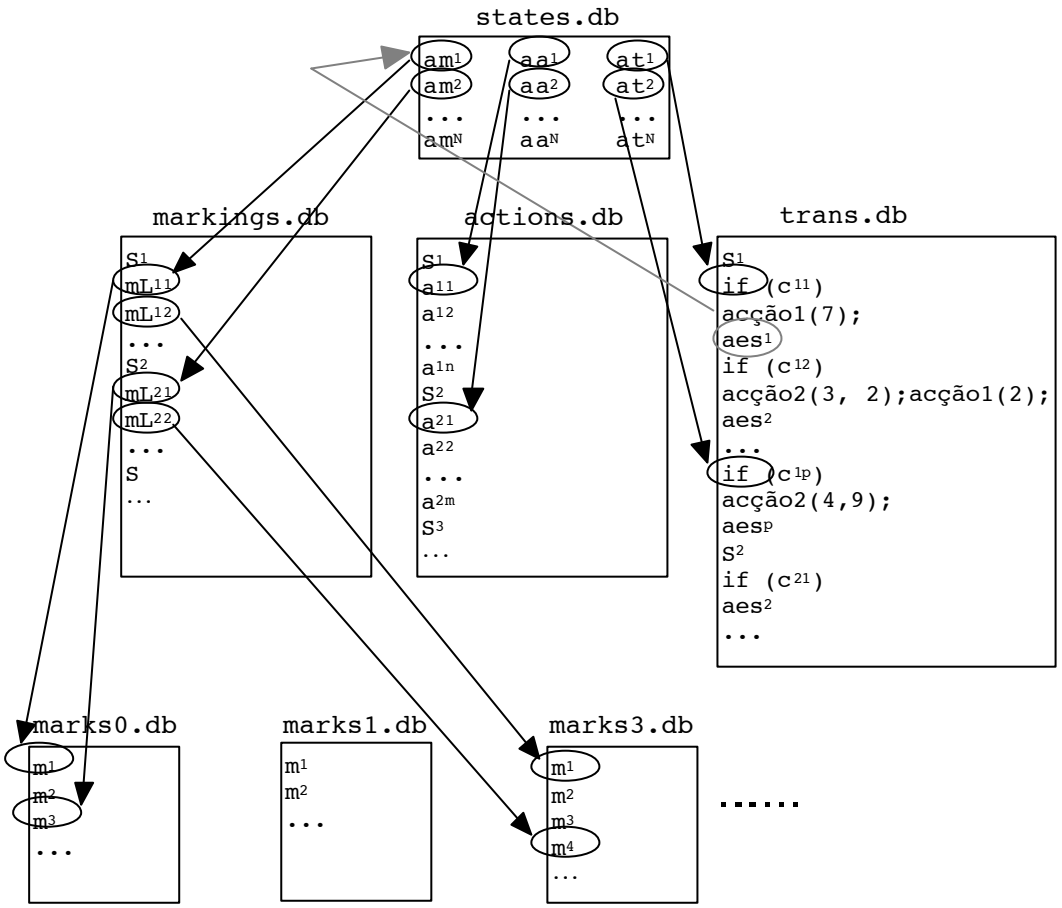


Figura 4.10 Estrutura de dados em memória secundária. Os ficheiros de suporte podem ser utilizados, posteriormente, por um interpretador da máquina de estados ou analisador do grafo de ocorrências. Ilustram-se apenas algumas das referências existentes entre os ficheiros.

4.5 Suporte para a rede sincronizada

A definição de uma rede sincronizada é feita com base na especificação de eventos associados às transições e acções associadas quer às transições quer aos lugares. As acções associadas às transições são invocadas aquando do disparo destas e como tal, podem ser função do vínculo de transição respectivo. As acções associadas aos lugares são invocadas antes de cada disparo da rede. Se pensarmos nas acções como simples afectações de variáveis de saída, as acções das

transições correspondem (na máquina de estados) a máquinas de Mealy e as acções associadas aos lugares a máquinas de Moore.

4.5.1 Eventos

Na especificação da rede, os eventos encontram-se sempre associados a transições. Consequentemente, o mesmo acontece na Biblioteca, em que cada objecto transição contem uma lista dos eventos que lhe estão associados.

Cada objecto *evento*, contêm uma lista de vínculos associados, pares *(variável, valor)*, a lista de transições a que está associado, o seu nome e uma variável *Booleana* para indicação do estado do evento aquando do disparo da rede para as várias combinações possíveis de valores dos eventos, com vista à construção da máquina de estados.

Conforme já anteriormente descrito em §2.1.4, os eventos que sejam função de variáveis da rede, do tipo inteiro, vão dar origem ao conjunto de eventos resultante da aplicação de todos as combinações de vínculos possíveis para o evento. Por exemplo, se um dado evento é definido como `ev(int a(2), int b(1))`, ele irá dar origem a seis eventos: ev<0,0>, ev<0,1>, ev<1, 0>, ev<1,1>, ev<2,0>, ev<2,1>. Ou seja, os eventos de alto-nível são decompostos em eventos de baixo-nível que se encontram associados a um só vínculo de transição. Esta estratégia possibilita uma mais simples geração das combinações de valores dos eventos, necessária para a construção da máquina de estados. Claro que poderíamos também "expandir" as transições com eventos deste tipo associados. Dessa forma passaria a existir sempre apenas um evento por transição. No entanto, tal não apresenta vantagens compensadoras em termo de implementação, pois embora simplificasse o algoritmo de disparo das transições (não seria necessário confrontar o vínculo da transição com cada um dos vínculos dos eventos), iria complicar demasiadamente a estrutura da rede, obrigando a criar novos arcos e transições. Ou seja, a rede transformar-se-ia, ainda que pontualmente, numa rede de mais baixo-nível. Outra consequência porventura ainda mais gravosa seria a perda de identidade entre a rede especificada e a rede analisada o que poderia dificultar em muito as tarefas de depuração do sistema modelado.

A "expansão" dos eventos permite também uma implementação simples para a escolha dos eventos

significativos em cada disparo da rede. Dado o conjunto dos vínculos das transições, um evento "significativo" é aquele que verifica a seguinte condição:

- O seu vínculo está contido em pelo menos um vínculo de uma transição a ele associada.

Como cada transição contem os seus eventos "expandidos", os eventos significativos são determinados testando simplesmente se o vínculo de cada evento está contido no vínculo da transição. Todos os eventos da transição que passem o teste são eventos significativos.

Se um evento não foi expandido, não apresenta vínculos pelo que a simples existência de um vínculo de transição é condição necessária e suficiente para a sua inclusão no conjunto dos eventos significativos.

Esta determinação dos eventos "significativos" pode resultar num substancial aumento de eficiência do algorimo de construção da máquina de estados descrito em [Gomes et al., 95:] visto implicar uma redução na quantidade de eventos a considerar para a geração de combinações.

4.5.2 Acções

Cada objecto *acção*, contêm uma lista de variáveis associados e o seu nome. Quando a acção é escrita no ficheiro `actions.db` as variáveis da acção encontram-se sempre vinculadas pelo que é possível escrever a chamada de função externa a que corresponde a acção. Por exemplo, se a acção se chamar *ligar* e apresentar como parâmetros duas variáveis inteiras, vinculadas com os valores 2 e 5 no momento de escrita no ficheiro, a chamada de função correspondente será: *ligar(2, 5)*.

Associadas às transições

A cada transição é possível associar um conjunto de acções. Estas são escritas no ficheiro `trans.db` aquando do disparo da transição. Nesse momento as variáveis da transição encontram-se vinculadas pelo que é seguro utilizá-las como parâmetros das acções.

Associadas aos lugares

A cada lugar é possível associar um conjunto de acções (vide §2.1.4). As acções de lugar são

executadas para todas as marcas presentes no lugar antes de cada disparo da rede. A execução das acções de lugar processa-se em duas fases:

1. Partindo da marcação do lugar, determinar combinações de vínculos possíveis para as variáveis de acção.
2. Para cada combinação de vínculos possível, avaliar a condição e se a condição for verdadeira, escrever chamada de função correspondente à acção e respectiva combinação de valores das suas variáveis, valores esses impostos pela combinação de vínculos.

A determinação das combinações de vínculos possíveis é idêntica à realizada para a determinação das instanciações possíveis para as variáveis dos arcos de entrada, quando utilizado o modificador *bypass_ttl* (vide §3.3.5). É importante notar que dado o cálculo deste vínculos ser realizado "entre-disparos", as variáveis da rede encontram-se disponíveis pelo que é possível utilizá-las como variáveis de acção. São estas mesmas variáveis que constituirão os possíveis parâmetros das funções externas correspondentes às acções. Como as acções são definidas antes dos lugares e das transições, qualquer acção pode vir a constituir quer uma acção de transição quer parte de uma acção de lugar. Neste caso terá obrigatóriamente uma condição associada.

4.6 Procedimentos de análise suportados

Conforme já referido, foram implementados dois procedimentos de análise: a construção do grafo de ocorrências de uma rede não-sincronizada e a construção da máquina de estados de uma rede sincronizada. Ambos recorrem à estrutura de dados já descrita.

4.6.1 Construção do grafo de ocorrências

O algoritmo de construção do grafo de ocorrências é apresentado em [Jensen 95](Figura 4.11). Trata-se de um algoritmo recursivo possibilitando vários tipos de percurso na árvore de recursividade: pré-ordem, pós-ordem, em-ordem ou nível-ordem. O pseudo-código em [Jensen 95] deixa estas opções em aberto dado tal não ser significativo.

```
Fila< Nó > espera;
Grafo< Nó, Arco > grafo;

pilha.Pôr( Nó(Marcação(M0) );
repete
   M1 = espera.Retirar();
   para cada elemento de vínculo b ε M1
  príncipio
      M2 = Seguinte(M1, b);
      espera.Pôr( Nó(M2) );
      grafo.AdicionarNó( Nó(M2) );
      grafo.AdicionarArco( Arco(M1, b, M2) );
   fim
até espera.Vazio();
```

Figura 4.11 Algoritmo de construção do grafo de ocorrências.

A função *Seguinte(M, b)* devolve a marcação resultante da aplicação do passo constituído unicamente pelo elemento de vínculo b, à marcação M.

Seguidamente apresenta-se de uma forma bastante mais detalhada, o pseudo-código para o algoritmo de construção do grafo de ocorrências da biblioteca. Importa notar que com o objectivo de não tornar a notação demasiado detalhada se omitiram a maior parte dos parâmetros de função.

```
GerarGrafoDeOcorrências() {
   BaseDeDados bd;
   bd.AdicionaNovoEstado(lugares); // guarda marcação inicial
   repete {
      bd.LêMarcação(lugares); //lê estado a processar da base de
dados
      AdicionaAoEstadoApontadorParaTransição(); // no states.db
      DeterminaVínculosDosConflitos()
      se (HáConflitosComVínculos()) {
         EscreveEstadoACL(); // sm.acl
         EscreveIdDeEstadoT(); // trans.db
         para cada conflito c {
            para cada transição apta ta {
               para cada vínculo de transição vt {
                  AplicaMarcaçãoDoEstadoLido();
                  DisparaTransiçãoComVínculo(ta, vt);
                  ActualizaMarcaçõesDosLugaresDeSaída(ta);
                  se (AindaNãoExisteMarcaçãoActual()) {
                     // adiciona marcação
                     bd.AdicionaNovoEstado(lugares);
                  }
                  //em trans.db
                  EscreveApontadorParaEstadoSeguinteT();
                  //em sm.acl
                  EscreveApontadorParaEstadoSeguinteACL();
               }
            }
         }
      senão { // não há transições para disparar
         se (TodosOsTTLsSãoZero()) { // Entalação
            EscreveEstadoTerminalACL(); // em sm.acl
         }
         senão {
            EscreveEstadoACL(); // sm.acl
            EscreveIdDeEstadoT(); // trans.db
            DecrementaTTLs();
            se (AindaNãoExisteMarcaçãoActual()) {
               // adiciona marcação
               bd.AdicionaNovoEstado(lugares);
            }
            //em trans.db
            EscreveApontadorParaEstadoSeguinteT();
            //em sm.acl
            EscreveApontadorParaEstadoSeguinteACL();
         }
      }
   } até NãoHaverEstadoSeguinte();
}
```

Quadro 4.6 Algoritmo de construção do grafo de ocorrências numa CpPNeTS não sincronizada.

O algoritmo efectua a construção do grafo em profundidade. Primeiro cria um nó correspondente à marcação inicial. Seguidamente, determina todas as marcações seguintes possíveis (através do disparo de todos os vínculos de transição). As marcações obtidas que ainda não constem no grafo vão constituir novos nós. Se não existem transições aptas e disponíveis, existem duas alternativas:

1. Decrementam os `ttls` criando desta forma um nova marcação. Se esta nova marcação não existe ainda no grafo então ela vai constituir um novo nó do grafo. Se já existir é criado um novo arco (apontador para o nó).

2. Os `ttls` são todos zero pelo que não se podem decrementar. Nesta situação é assinalado um nó terminal no ficheiro sm.acl, bem como, o respectivo arco. O ficheiro sm.acl é apresentado na secção §4.6.2.

No caso de uma rede temporizada, surgem nós cuja marcação apenas difere nos seus `ttls`. Tal deriva do facto dos próprios `ttls` constituirem parte integrante da marcação e não apenas um seu apêndice como sucede nas RdPCol [Jensen, 95: 145: 59]. Se considerarmos todos os nós que diferem apenas nos repectivos `ttls` como um único nó, verificar-se-á que o grafo dessa forma obtido é necessariamente um sub-grafo do obtido para a mesma rede mas sem temporizações associadas. Tal resulta de possíveis restrições adicionais impostas pela temporização da rede que a podem impedir de alcançar algumas das marcações da rede não-temporizada.

4.6.2 Construção da máquina de estados

A construção da máquina de estados síncrona a partir da definição de uma rede de Petri, constituiu a motivação fundamental para a definição das CpPNeTS, nomeadamente, para a inclusão de suporte ao sincronismo. O pseudo-código é idêntico ao utilizado para especificar o algoritmo de construção do grafo de ocorrências (Quadro 4.7).

A partir do conteúdo dos ficheiros da estrutura de dados em memória secundária, é possível extrair a máquina de estados propriamente dita: os estados são descritos pelo ficheiro `states.db`; as marcações encontram-se a partir dos ficheiros `markings.db` e `marks?.db` e as transições para os estados seguintes, acções associadas e respectivas combinações de eventos, podem encontrar-se no ficheiro `trans.db`. As acções associadas aos lugares são registadas num ficheiro próprio: o `actions.db`.

O ficheiro `actions.db` é perfeitamente legível mas o mesmo não se pode dizer dos restantes já citados. Como esses ficheiros se encontram formatados de forma a facilitar a leitura automática durante a criação da máquina de estados (ou do grafo de ocorrências), apresentam-se pouco legíveis. Como tal, optou-se pela criação de mais dois ficheiros: `sm.acl` e `states.txt`. Estes ficheiros são também criados durante a execução dos dois supra-citados algoritmos, mas apresentam um formato de fácil leitura, mantendo a possibilidade de um processamento automatico

simples.

```
GerarMáquinaDeEstados() {
   BaseDeDados bd;
   bd.AdicionaNovoEstado(lugares); // guarda marcação inicial
   repete {
      bd.LêMarcação(lugares); //lê estado a processar
      bd.AdicionaAcçõesDosLugares(lugares); // ao ficheiro
actions.db
      DeterminaVínculosDosConflitos();
      ApagaEventosSignificativos();
      se (HáConflitosComVínculos()) {
         EscreveEstadoACL(); // sm.acl
         EscreveIdDeEstadoT(); // trans.db
         se (QuantidadeDeEventosSignificativos > 0) {
            para cada combinação dos eventos significativos {
               PôeTodosOsEventosAFalso();

AtribuiValoresDaCombinaçãoAosEventosSignificativos();

               AplicaMarcaçãoDoEstadoLido();

               DisparaTodosOsConflitos();
               DecrementaTTLs();
               ActualizaMarcaçõesDosLugaresDeSaída();
               se (AindaNãoExisteMarcaçãoActual()) {
                  bd.AdicionaNovoEstado(lugares); //adiciona
marcação
               }
               EscreveApontadorParaEstadoSeguinteT();//em
trans.db
               EscreveApontadorParaEstadoSeguinteACL(); //em
sm.acl
            }
         }
         senão { // não há eventos significativos
            AplicarMarcaçãoDoEstadoLido()
            DisparaTodosOsConflitos();
            DecrementaTTLs();
            ActualizaMarcaçõesDosLugaresDeSaída();
            se (AindaNãoExisteNovaMarcação()) {
               bd.AdicionaNovoEstado(lugares); //adiciona
marcação
            }
            EscreveApontadorParaEstadoSeguinteT(); // em
trans.db
            EscreveApontadorParaEstadoSeguinteACL(); // em
sm.acl
         }
      }
      senão { // não há transições para disparar
         se (TodosOsTTLsSãoZero()) { // Entalação
            EscreverEstadoTerminalACL(); // em sm.acl
         }
         senão {
           EscreverEstadoACL(); // sm.acl
            EscreverIdDeEstadoT(); // trans.db
            DecrementaTTLs();
            ActualizaMarcaçõesDosLugaresDeSaída();
            se (AindaNãoExisteNovaMarcação()) {
               bd.AdicionaNovoEstado(lugares); //adiciona
marcação
            }
            EscreveApontadorParaEstadoSeguinteT(); // em
trans.db
            EscreveApontadorParaEstadoSeguinteACL(); // em
sm.acl
         }
      }
```

```
    } até NãoHaverEstadoSeguinte();
}
```

Quadro 4.7 Algoritmo de construção da máquina de estados de uma CpPNeTS sincronizada.

O ficheiro `sm.acl` é escrito numa linguagem proprietária para descrição de fluxogramas e estruturas em árvore e que foi utilizada para definição quer da máquina de estados quer do grafo de ocorrências. Essa linguagem pode ser interpretada por um programa de aplicação (allCLEAR^{TM}) para o ambiente MS-Windows de forma a obter a respectiva representação gráfica. Claro que a representação gráfica só é de facto legível para estruturas relativamente curtas. Apesar dessa natural limitação, é extremamente útil poder visualizar graficamente os grafos ou máquinas de estados obtidos, ainda que limitados pela dimensão dos mesmos. Por outro lado, o subconjunto da linguagem utilizado é extremamente simples, possibilitando uma fácil construção de outras ferramentas automáticas de processamento.

O ficheiro `states.txt` é constituído simplesmente pelas marcações de cada estado (ou nó) lugar a lugar num formato legível. É o complemento natural ao ficheiro `sm.acl`. Estes dois ficheiros descrevem totalmente o grafo de ocorrências. A máquina de estados, necessita ser complementada com o ficheiro `trans.db` caso existam acções associadas às transições e pelo ficheiro `actions.db` para as acções associadas aos lugares. Em particular, o gerador de código para PLCs ou um interpretador[81] da máquina de estados, necessitarão da informaçao contida nos ficheiros `states.db`, `trans.db` e `actions.db`.

É importante notar que os interpretadores da máquina de estado, não necessitam conhecer os ficheiros relativos às marcações da rede.

[81] Vide trabalho futuro (pág. 159).

Capítulo 5

Exemplos de aplicação

Os exemplos apresentados foram divididos em dois grupos principais: redes autónomas e redes não-autónomas. Para todas as redes, apresenta-se a especificação em CpPNeTS-DL. As palavras reservadas da linguagem CpPNeTS-DL surgem em negrito, e os identificadores de variáveis, tipos e funções da biblioteca em itálico. Para as redes autónomas apresenta-se ou o grafo de ocorrências, ou o resultado de uma execução da rede. Nas redes não-autónomas apresenta-se o grafo de ocorrências ou a máquina de estados gerada. Para não tornar a representação gráfica das redes demasiado pormenorizada, omitem-se muitas das notações. Estas constam na especificação em CpPNeTS-DL.

Para a representação gráfica dos grafos de ocorrências e das máquinas de estados, utilizou-se a já referida aplicação para MS-Windows™ 3.1, denominada allCLEAR™. Essa aplicação permite, entre outras coisas, a obtenção de uma representação gráfica de grafos a partir de um ficheiro contendo a sua descrição numa linguagem textual própria. Tanto o procedimento para geração do grafo de ocorrências, como o de geração da máquina de estados geram esse ficheiro. Desta forma, automatizou-se a obtenção da representação gráfica.

No Apêndice E encontra-se todo o código gerado pelo pré-processador, para cada um dos exemplos aqui referidos.

5.1 Redes autónomas

As redes autónomas não apresentam nem eventos nem acções, nem temporizações associadas às marcas. Seguem-se vários exemplos de redes deste tipo. Muitos dos exemplos são retirados da

literatura sobre RdPCol [Jensen, 92][Jensen, 95] como forma de possibilitar a comparação dos resultados obtidos.

5.1.1 Problema dos filósofos

Para o problema dos filósofos apresentam-se duas especificações: uma utilizando uma rede colorida e outra recorrendo a uma representação hierárquica de baixo-nível.

Rede colorida

A definição da rede colorida apresenta a nomenclatura utilizada em [Jensen, 95: 21] (Quadro 5.1). A representação gráfica desta rede é a que consta da Figura 1.6 onde é comparada com a correspondente rede de baixo-nível.

```
// exemplo in [Jensen, 95: 21]

net FiloCol

code {
private:
    typedef Int PH;
    typedef Int CS;
    enum { n = 5 };

    Marking Chopsticks(int p);
    MultiSet<Int> InitPlace();
}

variable<PH> ph;

place<PH> think "Thinking philosophers" { InitPlace() } [];
place<PH> eat   "Eating philosophers" {} [];
place<CS> chopsticks "Unused chopsticks" { InitPlace() } [];

transition take "Take Chopsticks" {}
   in
      think (ph) { return new MultiSet<PH>(ph); };
      chopsticks { return Chopsticks(ph); };
   out
      eat        { return new MultiSet<PH>(ph); };
   [] [] {
      cout << "philosopher " << ph << " took chopstiks" << endl;
   };

transition put "Put Down Chopsticks" {}
   in
      eat (ph) { return new MultiSet<PH>(ph); };
   out
      think      { return new MultiSet<PH>(ph); };
      chopsticks { return Chopsticks(ph); };
   [] [] {
      cout << "philosopher " << ph << " put chopstiks" << endl;
   };

code {
   MultiSet<Int>
   FiloCol::InitPlace() {
       MultiSet<Int> m;
       for(int i = 1; i <= n; i++) { m += MultiSet<Int>(Int(i));
}
       return m;
    }
    Marking
    FiloCol::Chopsticks(int p) {
       return new MultiSet<CS>(
                              MultiSet<CS>(Int(p)) +
                              MultiSet<CS>(Int((p == n) ? 1 : p +
1)));
    }
}
```

Quadro 5.1 Especificação, em CpPNeTS-DL, do problema dos filósofos (rede colorida).

Para alterar o número de filósofos na rede basta modificar a constante *n*. Utiliza-se a classe `Int` definida na biblioteca e correspondente a uma cor com um atributo do tipo inteiro. Apresenta-se,

seguidamente o grafo de ocorrências obtido (Figura 5.1).

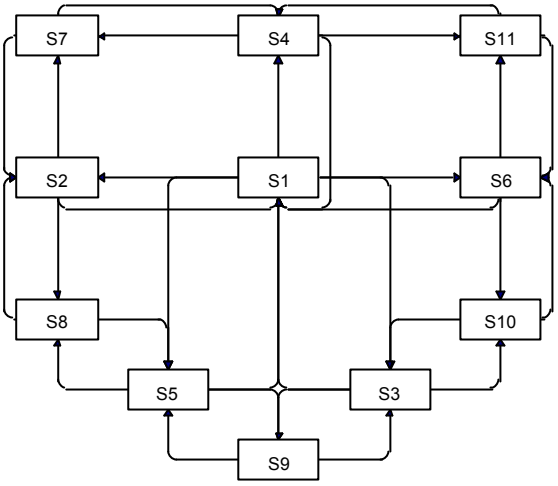


Figura 5.1 Grafo de ocorrências do problema dos filósofos. Os arcos a traço mais grosso saem do lugar S1.

O Quadro 5.2 especifica a marcação correspondente a cada nó do grafo, ou seja, o conteúdo do ficheiro `states.txt` criado juntamente com o grafo.

```
{ file automatically generated by
CpNetS v0.9 }

S1
think_1: 1#5 + 1#4 + 1#3 + 1#2 +
1#1
eat_1: empty
chopsticks_1: 1#5 + 1#4 + 1#3 +
1#2 + 1#1

S2
think_1: 1#4 + 1#3 + 1#2 + 1#1
eat_1: 1#5
chopsticks_1: 1#4 + 1#3 + 1#2

S3
think_1: 1#5 + 1#3 + 1#2 + 1#1
eat_1: 1#4
chopsticks_1: 1#3 + 1#2 + 1#1

S4
think_1: 1#5 + 1#4 + 1#2 + 1#1
eat_1: 1#3
chopsticks_1: 1#5 + 1#2 + 1#1

S5
think_1: 1#5 + 1#4 + 1#3 + 1#1
eat_1: 1#2
chopsticks_1: 1#5 + 1#4 + 1#1

S6
think_1: 1#5 + 1#4 + 1#3 + 1#2
eat_1: 1#1
chopsticks_1: 1#5 + 1#4 + 1#3

S7
think_1: 1#4 + 1#2 + 1#1
eat_1: 1#3 + 1#5
chopsticks_1: 1#2

S8
think_1: 1#4 + 1#3 + 1#1
eat_1: 1#2 + 1#5
chopsticks_1: 1#4

S9
think_1: 1#5 + 1#3 + 1#1
eat_1: 1#2 + 1#4
chopsticks_1: 1#1

S10
think_1: 1#5 + 1#3 + 1#2
eat_1: 1#1 + 1#4
chopsticks_1: 1#3

S11
think_1: 1#5 + 1#4 + 1#2
eat_1: 1#1 + 1#3
chopsticks_1: 1#5
```

Quadro 5.2 Marcação dos nós do grafo de ocorrências do problema dos filósofos.

Rede de baixo nível hierárquica

Contrariamente ao exemplo anterior, em que a compactação da rede é conseguida através de um maior complexidade da notação da rede, neste caso a compactação resulta da utilização do principio "dividir-para-reinar", ou seja, de uma decomposição da rede.

O grafo de ocorrências desta rede é, necessáriamente , igual ao do exemplo anterior. Como exemplo da possibilidade de programação de novas funcionalidades com base nos procedimentos da biblioteca já realizados, apresenta-se uma possível implementação de um procedimento de execução da rede (um *token-player* ou "jogador-de-marcas"). O procedimento é definido juntamente com a especificação da rede. Esta é constituída por duas sub-redes. Uma rede principal e uma macrotransição. A macrotransição representa um filósofo. A rede principal denominada `mesa`, contem quatro instâncias da macrotransição `filósofo`. Na Figura 1.9, foi já apresentada a representação gráfica desta rede.

```
// a mesa dos filosofos

net mesa
```

```
code {
public:
    void Simular(unsigned n);

private:
   void MostrarMesa(unsigned i);
}

place<Int> garfo1 "" { MS<Int>(Int(1)) } [];
place<Int> garfo2 "" { MS<Int>(Int(1)) } [];
place<Int> garfo3 "" { MS<Int>(Int(1)) } [];
place<Int> garfo4 "" { MS<Int>(Int(1)) } [];
place<Int> garfo5 "" { MS<Int>(Int(1)) } [];

macro transition filosofo f1(place garfo_direito = garfo1,
                             place garfo_esquerdo = garfo2);
macro transition filosofo f2(place garfo_direito = garfo2,
                             place garfo_esquerdo = garfo3);
macro transition filosofo f3(place garfo_direito = garfo3,
                             place garfo_esquerdo = garfo4);
macro transition filosofo f4(place garfo_direito = garfo4,
                             place garfo_esquerdo = garfo5);
macro transition filosofo f5(place garfo_direito = garfo5,
                             place garfo_esquerdo = garfo1);

code {

void
mesa::Simular(unsigned n) {
   for(unsigned i = 0; i < n; i++) {
      MostrarMesa(i);
      if (ComputeConflictsBindingsS()) {
         RandomFireConflicts();
         UpdateAllConflicts();
      }
   }
   MostrarMesa(n);
}
```

```
void
mesa::MostrarMesa(unsigned i) {
   cout << "\nM" << i;
   cout << "comendo:  " << !f1.comendo->GetMarking()->IsEmpty()
<<
                           !f2.comendo->GetMarking()->IsEmpty()
<<
                           !f3.comendo->GetMarking()->IsEmpty()
<<
                           !f4.comendo->GetMarking()->IsEmpty()
<<
                           !f5.comendo->GetMarking()->IsEmpty()
<< endl;
   cout << "pensando: " << !f1.pensando->GetMarking()->IsEmpty()
<<
                           !f2.pensando->GetMarking()->IsEmpty()
<<
                           !f3.pensando->GetMarking()->IsEmpty()
<<
                           !f4.pensando->GetMarking()->IsEmpty()
<<
                           !f5.pensando->GetMarking()->IsEmpty()
<< endl;
}

}
```

Quadro 5.3 A mesa dos filósofos.

```
// Um filosofo

net filosofo

code {
   static Expression Mark() {
      return new MS<Int>(Int(1));
   }
}

port place<Int> garfo_direito;
port place<Int> garfo_esquerdo;

place<Int> pensando "O filosofo pensa" { MS<Int>(Int(1)) } [];
place<Int> comendo "O filosofo come" {} [];

transition Comer "Retira garfos" {}
   in
      garfo_direito { return Mark(); };
      garfo_esquerdo { return Mark(); };
      pensando { return Mark(); };
   out
      comendo { return Mark(); };
   [] [] {
      cout << "O filosofo " << Inst() << " vai comer." << endl;
   };
```

```
transition Pensar "Repoe garfos" {}
   in
      comendo { return Mark(); };
   out
      pensando { return Mark(); };
      garfo_direito { return Mark(); };
      garfo_esquerdo { return Mark(); };
   [] [] {
      cout << "O filosofo " << Inst() << " vai pensar." << endl;
   };
```

Quadro 5.4 Macrotransição que modela um filósofo.

Note-se a utilização da constante 1 para especificação de uma marca de baixo-nível.

Para executar a simulação pretendida, basta invocar o procedimento respectivo na função main (Quadro 5.5).

```
// Code generated by pnetcpp V. 0.9, a Petri Net to C++ pre-
processor

#include "mesa.h"

mesa net;

int main() {
   net.Simular(15);
   return 0;
}
```

Quadro 5.5 Invocação do procedimento de simulação.

Como resultado obtemos as mensagens correspondentes ao disparo das transições, especificadas nos segmentos de código das transições, e as marcações dos vários lugares da rede. Estas marcações são especificadas pelos valores 0 ou 1 para indicar a presença da marca. Tal é suficiente , por se tratar de uma rede binária.

```
M0
comendo:  00000
garfos:   11111
pensando: 11111

O filosofo 4 vai comer.

M1
comendo:  00010
garfos:   11100
pensando: 11101

O filosofo 2 vai comer.
O filosofo 4 vai
pensar.

M2
comendo:  01000
garfos:   10011
pensando: 10111

O filosofo 5 vai comer.
O filosofo 2 vai
pensar.

M3
comendo:  00001
garfos:   01110
pensando: 11110

O filosofo 2 vai comer.
O filosofo 5 vai
pensar.

M4
comendo:  01000
garfos:   10011
pensando: 10111

O filosofo 5 vai comer.
O filosofo 2 vai
pensar.

M5
comendo:  00001
garfos:   01110
pensando: 11110

O filosofo 3 vai comer.
O filosofo 5 vai
pensar.

M6
comendo:  00100
garfos:   11001
pensando: 11011

O filosofo 5 vai comer.
```

```
O filosofo 3 vai
pensar.

M7
comendo:  00001
garfos:   01110
pensando: 11110

O filosofo 3 vai comer.
O filosofo 5 vai
pensar.

M8
comendo:  00100
garfos:   11001
pensando: 11011

O filosofo 1 vai comer.
O filosofo 3 vai
pensar.

M9
comendo:  10000
garfos:   00111
pensando: 01111

O filosofo 3 vai comer.
O filosofo 1 vai
pensar.

M10
comendo:  00100
garfos:   11001
pensando: 11011

O filosofo 1 vai comer.
O filosofo 3 vai
pensar.

M11
comendo:  10000
garfos:   00111
pensando: 01111

O filosofo 4 vai comer.
O filosofo 1 vai
pensar.

M12
comendo:  00010
garfos:   11100
pensando: 11101

O filosofo 1 vai comer.
O filosofo 4 vai
pensar.

M13
comendo:  10000
garfos:   00111
pensando: 01111

O filosofo 3 vai comer.
O filosofo 1 vai
pensar.

M14
comendo:  00100
garfos:   11001
pensando: 11011

O filosofo 1 vai comer.
O filosofo 3 vai
pensar.
```

Quadro 5.6 Execução do problema dos filósofos.

É importante notar que nunca ocorrem *deadlocks*. Tal resulta da modelação implementada exigir que cada filósofo retire ou coloque ambos os garfos numa operação indivisível.. Note-se que a ordem de disparo das transições é aleatória mas disparando o máximo de transições habilitadas.

5.1.2 O problema dos filósofos cooperantes

O problema dos filósofos cooperantes consiste numa variação simples do problema dos filósofos, com o objectivo de ilustrar mais algumas das potencialidades das CpPNeTS e do sistema desenvolvido.

Neste exemplo, cada filósofo pode retirar apenas um garfo de cada vez. Fica então à espera de retirar o seu segundo garfo. Se passado um dado tempo não o consegue, volta a colocar na mesa o garfo que possui. Seguidamente esse mesmo garfo poderá ser retirado por ele ou por qualquer um dos restantes filósofos. Logo que um filósofo possui os dois garfos passa a comer durante um determinado tempo e não pensa sequer em pousar os garfos. Quando acaba de comer pousa os dois garfos simultaneamente. Como estes filósofos gostam mais de comer do que de pensar, só pensam enquanto não conseguem obter os dois garfos. Para modelar este sistema é necessario recorrer a duas capacidades das CpPNeTS que não foram aplicadas nos exemplos anteriores: temporizações associadas às marcas, e prioridades associadas às transições. A rede é a que se

apresenta na Figura 5.2. As marcas representadas correspondem a marcas coloridas.

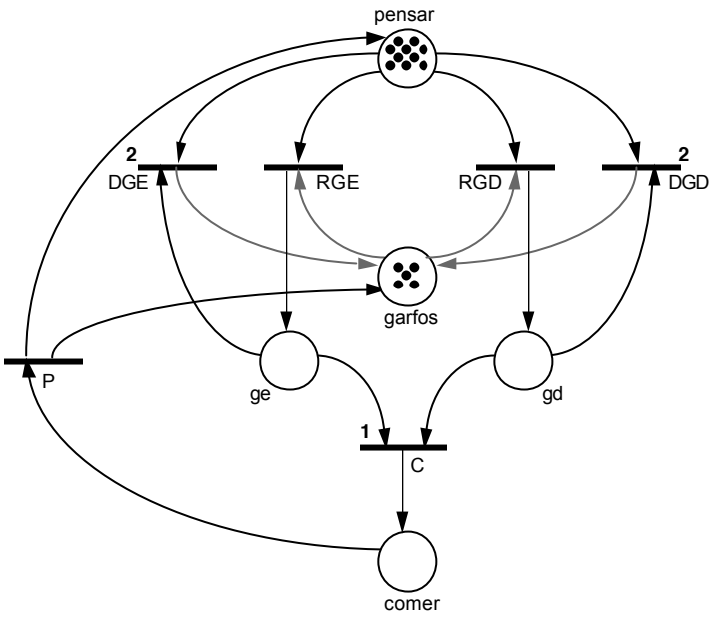


Figura 5.2 Filósofos cooperantes. Os nomes das transições e lugares são os da especificação em CpPNeTS-DL. Os números junto às transições DGE, DGD e C são as prioridades associadas. Menor número significa maior prioridade

Segue-se a especificação em CpPNeTS-DL:

```
. // O problema dos filosofos cooperantes

net FiloCoop

code {
public:
    void Simular(unsigned n);
    void AtribuirTempos(unsigned e, unsigned d, unsigned c) {
       tempoComGarfoEsquerdo = e;
       tempoComGarfoDireito  = d;
       tempoAComer           = c;
    }

private:
    unsigned tempoComGarfoEsquerdo;
    unsigned tempoComGarfoDireito;
    unsigned tempoAComer;

    class TFiloGarfo : public TColor {
    public:
       Int f;
       Int g;
       TFiloGarfo() {}
       TFiloGarfo(int ff, int gg, int t = 0)
       : TColor(t), f(ff), g(gg) {}
       friend bool operator==(const TFiloGarfo& esq,
                              const TFiloGarfo& dir) {
          return (esq.f == dir.f) && (esq.g == dir.g) &&
                 (esq.ttl == dir.ttl);
       }
       friend bool operator!=(const TFiloGarfo& esq,
```

```
                                const TFiloGarfo& dir) {
            return (esq.f != dir.f) || (esq.g != dir.g) ||
                   (esq.ttl != dir.ttl);
         }
         void Write(ostream& s) const {
            s << f << " " << g << " " << ttl;
         }
         void Read(istream& s) {
            s >> f >> g >> ttl;
         }
      };

      typedef TInt Filosofo;
      typedef Int  Garfo;
      enum { n = 5 };

      Int GE(int f) const { return (f == 1) ? n : f - 1; }
      Int GD(int f) const { return f; }
      Int FE(int g) const { return g; }
      Int FD(int g) const { return (g == n) ? 1 : g + 1; }

      MS<Filosofo> InicializarF();
      MS<Garfo> InicializarG();

      void MostrarMesa(int i);
}

variable<Filosofo> f;
variable<TFiloGarfo> fg1;
variable<TFiloGarfo> fg2;

place<Filosofo> pensar "Filosofos a pensar" { InicializarF() }
[];
place<Filosofo> comer  "Filosofos a comer" {} [];
place<Garfo> garfos "Garfos disponiveis" { InicializarG() } [];

place<TFiloGarfo> ge "Filosofo com garfo esquerdo na mao para
comer" {} [];
place<TFiloGarfo> gd "Filosofo com garfo direito na mao para
comer" {} [];

transition RGE "Retirar garfo esquerdo" {}
   in
      pensar (f) { return new MS<Filosofo>(f); };
      garfos { return new MS<Garfo>(Garfo(GE(f.t))); };
   out
      ge    { return new MS<TFiloGarfo>(TFiloGarfo(f.t, GE(f.t),
                                        tempoComGarfoEsquerdo));
            };
   [] [] {
      cout << "filosofo " << f.t << " retirou garfo esquerdo" <<
endl;
   };

transition RGD "Retirar garfo direito" {}
   in
      pensar (f) { return new MS<Filosofo>(f); };
      garfos { return new MS<Garfo>(Garfo(GD(f.t))); };
   out
      gd    { return new MS<TFiloGarfo>(TFiloGarfo(f.t, GD(f.t),
                                        tempoComGarfoDireito));
            };
   [] [] {
      cout << "filosofo " << f.t << " retirou garfo direito" <<
endl;
   };
```

```
transition C "Vai comer" {
   return (fg1.f == fg2.f) && (FD(fg1.g) == fg2.g) ; }
   in
      ge (bypass_ttl fg1) { return new MS<TFiloGarfo>(fg1); };
      gd (bypass_ttl fg2) { return new MS<TFiloGarfo>(fg2); };
   out
      comer { return new MS<Filosofo>(Filosofo(FD(fg1.g),
                                       fg1.f + tempoAComer)); };
   [] [] {
      cout << "filosofo " << FD(fg1.g) << " vai comer" << endl;
   } priority = 1;

transition P "Vai pensar" {}
   in
      comer (f) { return new MS<Filosofo>(f); }
   out
      pensar { return new MS<Filosofo>(Filosofo(f.t, 0), 2); };
      garfos { return new MS<Garfo>( MS<Garfo>(GD(f.t)) +
                                     MS<Garfo>(GE(f.t)) );
             };
   [] [] {
      cout << "filosofo " << f.t << " vai pensar" << endl;
   };

transition DGE "Desiste do garfo esquerdo" {}
   in
      ge (fg1) { return new MS<TFiloGarfo>(fg1); }
   out
      pensar { return new MS<Filosofo>(Filosofo(fg1.f, 0)); };
      garfos { return new MS<Garfo>(MS<Garfo>(fg1.g)); }
   [] [] {
      cout << "filosofo " << fg1.f <<
              " desistiu do garfo esquerdo" << endl;
   } priority = 2;

transition DGD "Desiste do garfo direito" {}
   in
      gd (fg1) { return new MS<TFiloGarfo>(fg1); }
   out
      pensar { return new MS<Filosofo>(Filosofo(fg1.f, 0)); };
      garfos { return new MS<Garfo>(MS<Garfo>(fg1.g)); }
   [] [] {
      cout << "filosofo " << fg1.f <<
              " desistiu do garfo direito" << endl;
   } priority = 2;

code {
   MS<FiloCoop::Filosofo>
   FiloCoop::InicializarF() {
       MS<Filosofo> m;
       for(int i = 1; i <= n; i++) {
          m += 2 * MS<Filosofo>(Filosofo(i, 0));
       }
       return m;
   }

   MS<FiloCoop::Garfo>
   FiloCoop::InicializarG() {
       MS<Garfo> m;
       for(int i = 1; i <= n; i++) { m += MS<Garfo>(Garfo(i)); }
       return m;
   }

void
FiloCoop::Simular(unsigned n) {
```

```
   for(unsigned i = 0; i < n; i++) {
      MostrarMesa(i);
      if (ComputeConflictsBindingsS()) {
         RandomFireConflicts();
         DecrementTTLs();
         UpdateAllConflicts();
      }
      else if (TTLsAreAllZero()) {
         cout << "Deadlock" << endl;
         return;
      }
      else {
         DecrementTTLs();
      }
   }
   MostrarMesa(n);
}

void
FiloCoop::MostrarMesa(int i) {
   cout << "\nMarcacao " << i << "\npensando:  " <<
           *pensar->GetMarking();
   cout << "\ncom garfo esquerdo " << *ge->GetMarking();
   cout << "\ncom garfo direito " << *gd->GetMarking();
   cout << "\ngarfos na mesa " << *garfos->GetMarking();
   cout << "\ncomendo: " << *comer->GetMarking() <<  "\n" <<
endl;
}

}
```

Figura 5.3 Filósofos cooperantes em CpPNeTS-DL. Note-se a utilização do modificador `bypass_ttl` nas variáveis dos arcos de entrada da transição `c`.

O lugar pensar contem duas marcas por filósofo. Para obter cada um dos garfos é necessária uma dessas marcas. Desta forma é possível considerar que um filósofo está a pensar enquanto existir pelo menos uma marca no lugar pensar. Outra hipótese será considerar que quando o filósofo tem um garfo na mão já não pensa, apesar de também não comer (está simplesmente a consumir tempo). Quando um filósofo obtem um dos garfos é iniciado um temporizador associado à marca contida no lugar `ge` ou `gd` conforme se trate do garfo esquerdo ou do garfo direito, respectivamente. Se esse temporizador chega a zero e o filósofo ainda não conseguiu obter o segundo garfo, então a transição `DGE` (ou `DGD`) repõe o garfo, no lugar `garfos`, e a marca correspondente ao filósofo no lugar `pensar`. Logo que um filósofo consegue obter o segundo garfo, a transição `c` é disparada prioritariamente às outras duas transições da mesma contenda (`DGE` e `DGD`) independentemente dos valores das temporizações das marcas respectivas em `ge` ou `gd`. Para obter esta semântica são necessários dois cuidados:

1. A transição `c` possui uma prioridade maior (menor valor) do que qualquer uma das transições `DGE` ou `DGD`. Desta forma, um filósofo que possua os dois garfos, irá sempre comer, em vez de depositar um dos garfos.

2. As variáveis dos arcos de entrada da transição `c` são declaradas como instanciáveis com todas as marcas presentes no respectivo lugar de entrada, independentemente do valor dos seus `ttls`. Para tal utiliza-se o modificador `bypass_ttl`. A guarda verifica se atributos da cor, com excepção dos `ttls`, são iguais em ambas as variáveis. Note-se que não é possível utilizar apenas uma variável de arco num dos arcos, com o outro arco a utilizá-la como valor-direito. Tal obrigaria a uma igualdade também dos atributos `ttls` das marcas presentes nos lugares, o que, neste caso, não é desejável.

Cada filósofo pensa durante uma quantidade de tempo igual ao seu numero de série mais um parâmetro fornecido em tempo de execução. Também os tempos de permanência com o garfo na mão são dados em tempo de execução, na linha de comando da *shell* do sistema operativo (vide ficheiro `main.c` no Anexo A).

No quadro seguinte apresenta-se o resultado da execução da rede com os tempos indicados na respectiva legenda.

```
Marcacao 0
pensando:  2#5 0 + 2#4 0 + 2#3 0 +
2#2 0 + 2#1 0
com garfo esquerdo empty
com garfo direito empty
garfos na mesa 1#5 + 1#4 + 1#3 + 1#2
+ 1#1
comendo: empty

filosofo 1 retirou garfo direito

Marcacao 1
pensando:  2#5 0 + 2#4 0 + 2#3 0 +
2#2 0 + 1#1 0
com garfo esquerdo empty
com garfo direito 1#1 1 3
garfos na mesa 1#5 + 1#4 + 1#3 + 1#2
comendo: empty

filosofo 4 retirou garfo direito

Marcacao 2
pensando:  2#5 0 + 1#4 0 + 2#3 0 +
2#2 0 + 1#1 0
com garfo esquerdo empty
com garfo direito 1#4 4 3 + 1#1 1 2
garfos na mesa 1#5 + 1#3 + 1#2
comendo: empty

filosofo 3 retirou garfo direito

Marcacao 3
pensando:  2#5 0 + 1#4 0 + 1#3 0 +
2#2 0 + 1#1 0
com garfo esquerdo empty
com garfo direito 1#3 3 3 + 1#4 4 2 +
1#1 1 1
garfos na mesa 1#5 + 1#2
comendo: empty

filosofo 3 retirou garfo esquerdo

Marcacao 4
pensando:  2#5 0 + 1#4 0 + 2#2 0 +
1#1 0
com garfo esquerdo 1#3 2 3
com garfo direito 1#3 3 2 + 1#4 4 1 +
1#1 1 0
garfos na mesa 1#5
comendo: empty

filosofo 5 retirou garfo direito
filosofo 3 vai comer

Marcacao 5
pensando:  1#5 0 + 1#4 0 + 2#2 0 +
1#1 0
com garfo esquerdo empty
com garfo direito 1#5 5 3 + 1#4 4 0 +
1#1 1 0
garfos na mesa empty
comendo: 1#3 5

filosofo 4 desistiu do garfo direito

Marcacao 6
pensando:  1#5 0 + 2#4 0 + 2#2 0 +
1#1 0
com garfo esquerdo empty
com garfo direito 1#5 5 2 + 1#1 1 0
garfos na mesa 1#4
```

```
comendo: 1#3 4

filosofo 5 retirou garfo esquerdo
filosofo 1 desistiu do garfo direito

Marcacao 7
pensando:  2#4 0 + 2#2 0 + 2#1 0
com garfo esquerdo 1#5 4 3
com garfo direito 1#5 5 1
garfos na mesa 1#1
comendo: 1#3 3

filosofo 1 retirou garfo direito
filosofo 5 vai comer

Marcacao 8
pensando:  2#4 0 + 2#2 0 + 1#1 0
com garfo esquerdo empty
com garfo direito 1#1 1 3
garfos na mesa empty
comendo: 1#5 7 + 1#3 2


Marcacao 9
pensando:  2#4 0 + 2#2 0 + 1#1 0
com garfo esquerdo empty
com garfo direito 1#1 1 2
garfos na mesa empty
comendo: 1#5 6 + 1#3 1


Marcacao 10
pensando:  2#4 0 + 2#2 0 + 1#1 0
com garfo esquerdo empty
com garfo direito 1#1 1 1
garfos na mesa empty
comendo: 1#5 5 + 1#3 0

filosofo 3 vai pensar

Marcacao 11
pensando:  2#3 0 + 2#4 0 + 2#2 0 +
1#1 0
com garfo esquerdo empty
com garfo direito 1#1 1 0
garfos na mesa 1#3 + 1#2
comendo: 1#5 4

filosofo 3 retirou garfo esquerdo
filosofo 1 desistiu do garfo direito

Marcacao 12
pensando:  1#3 0 + 2#4 0 + 2#2 0 +
2#1 0
com garfo esquerdo 1#3 2 3
com garfo direito empty
garfos na mesa 1#1 + 1#3
comendo: 1#5 3

filosofo 2 retirou garfo esquerdo

Marcacao 13
pensando:  1#3 0 + 2#4 0 + 1#2 0 +
2#1 0
com garfo esquerdo 1#2 1 3 + 1#3 2 2
com garfo direito empty
garfos na mesa 1#3
comendo: 1#5 2

filosofo 4 retirou garfo esquerdo

Marcacao 14
pensando:  1#3 0 + 1#4 0 + 1#2 0 +
2#1 0
com garfo esquerdo 1#4 3 3 + 1#2 1 2
+ 1#3 2 1
com garfo direito empty
garfos na mesa empty
comendo: 1#5 1


Marcacao 15
pensando:  1#3 0 + 1#4 0 + 1#2 0 +
2#1 0
com garfo esquerdo 1#4 3 2 + 1#2 1 1
+ 1#3 2 0
com garfo direito empty
garfos na mesa empty
comendo: 1#5 0

filosofo 3 desistiu do garfo esquerdo
filosofo 5 vai pensar

Marcacao 16
pensando:  2#5 0 + 2#3 0 + 1#4 0 +
1#2 0 + 2#1 0
com garfo esquerdo 1#4 3 1 + 1#2 1 0
com garfo direito empty
garfos na mesa 1#5 + 1#4 + 1#2
comendo: empty

filosofo 3 retirou garfo esquerdo
filosofo 2 desistiu do garfo esquerdo

Marcacao 17
pensando:  2#5 0 + 1#3 0 + 1#4 0 +
2#2 0 + 2#1 0
com garfo esquerdo 1#3 2 3 + 1#4 3 0
com garfo direito empty
garfos na mesa 1#1 + 1#5 + 1#4
comendo: empty

filosofo 5 retirou garfo esquerdo
filosofo 4 desistiu do garfo esquerdo

Marcacao 18
pensando:  1#5 0 + 1#3 0 + 2#4 0 +
2#2 0 + 2#1 0
com garfo esquerdo 1#5 4 3 + 1#3 2 2
com garfo direito empty
garfos na mesa 1#3 + 1#1 + 1#5
comendo: empty

filosofo 4 retirou garfo esquerdo

Marcacao 19
pensando:  1#5 0 + 1#3 0 + 1#4 0 +
2#2 0 + 2#1 0
com garfo esquerdo 1#4 3 3 + 1#5 4 2
+ 1#3 2 1
com garfo direito empty
garfos na mesa 1#1 + 1#5
comendo: empty

filosofo 1 retirou garfo esquerdo
```

```
Marcacao 20
pensando:  1#5 0 + 1#3 0 + 1#4 0 +
2#2 0 + 1#1 0
com garfo esquerdo 1#1 5 3 + 1#4 3 2
+ 1#5 4 1 + 1#3 2 0
com garfo direito empty
garfos na mesa 1#1
comendo: empty

filosofo 2 retirou garfo esquerdo
filosofo 3 desistiu do garfo esquerdo

Marcacao 21
pensando:  1#5 0 + 2#3 0 + 1#4 0 +
1#2 0 + 1#1 0
com garfo esquerdo 1#2 1 3 + 1#1 5 2
+ 1#4 3 1 + 1#5 4 0
com garfo direito empty
garfos na mesa 1#2
comendo: empty

filosofo 2 retirou garfo direito
filosofo 5 desistiu do garfo esquerdo

Marcacao 22
pensando:  2#5 0 + 2#3 0 + 1#4 0 +
1#1 0
com garfo esquerdo 1#2 1 2 + 1#1 5 1
+ 1#4 3 0
com garfo direito 1#2 2 3
garfos na mesa 1#4
comendo: empty

filosofo 4 retirou garfo direito
filosofo 2 vai comer

Marcacao 23
pensando:  2#5 0 + 2#3 0 + 1#1 0
com garfo esquerdo 1#1 5 0 + 1#4 3 0
com garfo direito 1#4 4 3
garfos na mesa empty
comendo: 1#2 4

filosofo 4 vai comer

Marcacao 24
pensando:  2#5 0 + 2#3 0 + 1#1 0
com garfo esquerdo 1#1 5 0
com garfo direito empty
garfos na mesa empty
comendo: 1#4 6 + 1#2 3

filosofo 1 desistiu do garfo esquerdo
```

```
Marcacao 25
pensando:  2#5 0 + 2#3 0 + 2#1 0
com garfo esquerdo empty
com garfo direito empty
garfos na mesa 1#5
comendo: 1#4 5 + 1#2 2

filosofo 5 retirou garfo direito

Marcacao 26
pensando:  1#5 0 + 2#3 0 + 2#1 0
com garfo esquerdo empty
com garfo direito 1#5 5 3
garfos na mesa empty
comendo: 1#4 4 + 1#2 1


Marcacao 27
pensando:  1#5 0 + 2#3 0 + 2#1 0
com garfo esquerdo empty
com garfo direito 1#5 5 2
garfos na mesa empty
comendo: 1#4 3 + 1#2 0

filosofo 2 vai pensar

Marcacao 28
pensando:  2#2 0 + 1#5 0 + 2#3 0 +
2#1 0
com garfo esquerdo empty
com garfo direito 1#5 5 1
garfos na mesa 1#2 + 1#1
comendo: 1#4 2

filosofo 2 retirou garfo direito

Marcacao 29
pensando:  1#2 0 + 1#5 0 + 2#3 0 +
2#1 0
com garfo esquerdo empty
com garfo direito 1#2 2 3 + 1#5 5 0
garfos na mesa 1#1
comendo: 1#4 1

filosofo 1 retirou garfo direito
filosofo 5 desistiu do garfo direito

Marcacao 30
pensando:  1#2 0 + 2#5 0 + 2#3 0 +
1#1 0
com garfo esquerdo empty
com garfo direito 1#1 1 3 + 1#2 2 2
garfos na mesa 1#5
comendo: 1#4 0
```

Quadro 5.7 Resultado da simulação dos problema dos filósofos cooperantes. Os tempos de permanência com os garfos na mão são de 2 e o tempo a comer é 2 mais o número de série do filósofo. Por exemplo, o filósofo 3 tem um `ttl` de 5 quando começa a comer. Foram efectuadas 30 iterações.

5.1.3 Sistema de Reserva de Recursos (*Resource Allocation System*)

Este exemplo foi retirado de [Jensen, 95: 2]. Trata-se de uma versão simplificada do exemplo em [Jensen, 92: 14]. A simplificação consistiu na omissão dos contadores de ciclo de forma a tornar finito o grafo de ocorrências da rede.

O exemplo assume a existência de um conjunto de processos. Existem dosi tipos de processos (denominados processos-p e processos-q) e três tipos de recursos (recursos-r, recursos-s e recursos-t). Cada processo é ciclico, e durante cada parte do seu ciclo necessita ter acesso exclusivo a quantidades variáveis de recursos. Na rede da Figura 5.4, o ciclo dos processos-p é a→b→c→d→e→a e dos processos-q é b→c→d→e→b.

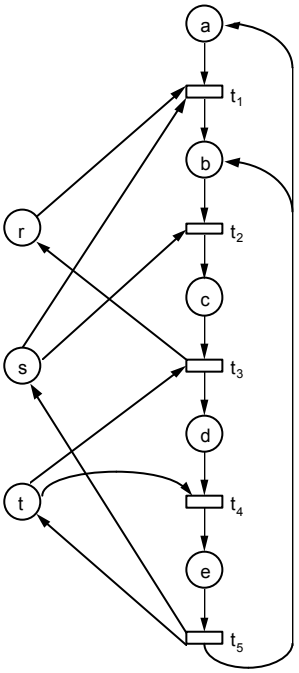


Figura 5.4 Sistema de reserva de recursos.

Apresentam-se as implementações de duas versões desse sistema: o já citado e outro a que se adicionaram temporizações e que se pode encontrar em [Jensen, 95: 147]. As temporizações das marcações iniciais foram omitidas dado serem irrelevantes por apresentarem valores menores ou iguais que o tempo actual aí considerado: 641.

Sem temporizações

```
// rede em [Jensen, 92: 10] simplificada conforme [Jensen, 92:
142]

code {
   const int p = 0;
   const int q = 1;
}

net AllocSys

code {
   typedef Int U;
   typedef Int E;
}

variable<U> x;

place<U> a "a" { 3 * MultiSet<U>(U(q)) } [];
place<U> b "b" { 2 * MultiSet<U>(U(p)) } [];
place<U> c "c" {} [];
place<U> d "d" {} [];
place<U> e "e" {} [];

place<E> r "r" { MultiSet<E>(E(1)) } [];
place<E> s "s" { 3 * MultiSet<E>(E(1)) } [];
place<E> t "t" { 2 * MultiSet<E>(E(1)) } [];


transition t1 "t1" { return x == q; }
   in
      a (x) { return new MultiSet<U>(U(x)); };
      r     { return new MultiSet<E>(E(1)); };
      s     { return new MultiSet<E>(E(1)); };
   out
      b     { return new MultiSet<U>(U(x)); };
   [] [] {
      cout << "t1 disparou" << endl;
   };

transition t2 "t2" {}
   in
      b (x) { return new MultiSet<U>(U(x)); };
      s     { return new MultiSet<E>(E(1), 1 + !!(x == p)); };
   out
      c     { return new MultiSet<U>(U(x)); };
   [] [] {
      cout << "t2 disparou" << endl;
   };

transition t3 "t3" {}
   in
      c (x) { return new MultiSet<U>(U(x)); };
      t     { if (x == p) return new MultiSet<E>(E(1));
              else        return new MultiSet<E>(); };
   out
      r     { if (x == q) return new MultiSet<E>(E(1));
              else        return new MultiSet<E>(); };
      d     { return new MultiSet<U>(U(x)); };
   [] [] {
      cout << "t3 disparou" << endl;
   };
```

```
transition t4 "t4" {}
   in
      d (x) { return new MultiSet<U>(U(x)); };
      t     { return new MultiSet<E>(E(1)); };
   out
      e     { return new MultiSet<U>(U(x)); };
   [] [] {
      cout << "t4 disparou" << endl;
   };

transition t5 "t5" {}
   in
      e (x) { return new MultiSet<U>(U(x)); };
   out
      s     { return new MultiSet<E>(E(1), 2); };
      t     { return new MultiSet<E>(E(1), 1 + !!(x == p)); };
      a     { if (x == q) return new MultiSet<U>(U(q));
              else        return new MultiSet<U>();  };
      b     { if (x == p) return new MultiSet<U>(U(p));
              else        return new MultiSet<U>();  };
   [] [] {
      cout << "t5 disparou" << endl;
   };
```

Quadro 5.8 Especificação, em CpPNeTS-DL, do sistema de reserva de recursos sem temporizações.

Seguidamente apresenta-se o grafo de ocorrências obtido e o ficheiro gerado (`states.txt`) que contêm a marcação de cada um dos seus nós.

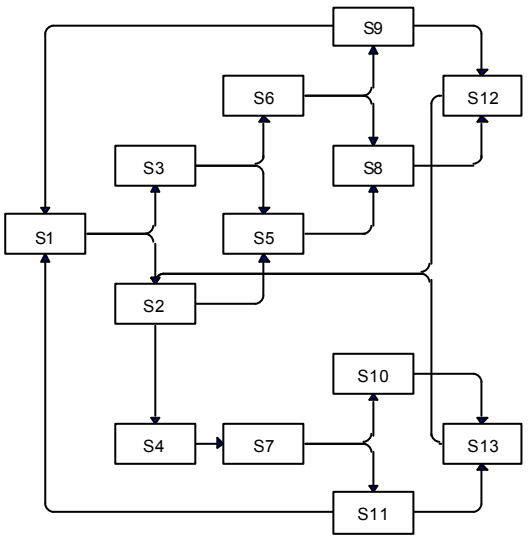


Figura 5.5 Grafo de ocorrências do sistema de reserva, sem temporizações.

```
{ file
automatical
ly
generated
by CpPNeTS-
Lib v1.0 }


S1
a_1: 3#1
b_1: 2#0
c_1: empty
d_1: empty
e_1: empty
r_1: 1#1
s_1: 3#1
t_1: 2#1

S2
a_1: 2#1
b_1: 1#1 +
2#0
c_1: empty
d_1: empty
e_1: empty
r_1: empty
s_1: 2#1
t_1: 2#1
```

```
S3
a_1: 3#1
b_1: 1#0
c_1: 1#0
d_1: empty
e_1: empty
r_1: 1#1
s_1: 1#1
t_1: 2#1

S4
a_1: 2#1
b_1: 2#0
c_1: 1#1
d_1: empty
e_1: empty
r_1: empty
s_1: 1#1
t_1: 2#1

S5
a_1: 2#1
b_1: 1#1 +
1#0
c_1: 1#0
d_1: empty
e_1: empty
r_1: empty
s_1: empty
```

```
t_1: 2#1

S6
a_1: 3#1
b_1: 1#0
c_1: empty
d_1: 1#0
e_1: empty
r_1: 1#1
s_1: 1#1
t_1: 1#1

S7
a_1: 2#1
b_1: 2#0
c_1: empty
d_1: 1#1
e_1: empty
r_1: 1#1
s_1: 1#1
t_1: 2#1

S8
a_1: 2#1
b_1: 1#1 +
1#0
c_1: empty
d_1: 1#0
e_1: empty
```

```
r_1: empty
s_1: empty
t_1: 1#1

S9
a_1: 3#1
b_1: 1#0
c_1: empty
d_1: empty
e_1: 1#0
r_1: 1#1
s_1: 1#1
t_1: empty

S10
a_1: 1#1
b_1: 1#1 +
2#0
c_1: empty
d_1: 1#1
e_1: empty
r_1: empty
s_1: empty
t_1: 2#1

S11
a_1: 2#1
b_1: 2#0
c_1: empty
```

```
d_1: empty
e_1: 1#1
r_1: 1#1
s_1: 1#1
t_1: 1#1

S12
a_1: 2#1
b_1: 1#1 +
1#0
c_1: empty
d_1: empty
e_1: 1#0
r_1: empty
s_1: empty
t_1: empty

S13
a_1: 1#1
b_1: 1#1 +
2#0
c_1: empty
d_1: empty
e_1: 1#1
r_1: empty
s_1: empty
t_1: 1#1
```

Quadro 5.9 Marcação dos nós do grafo de ocorrências do sistema de reserva sem temporizações.

Com temporizações

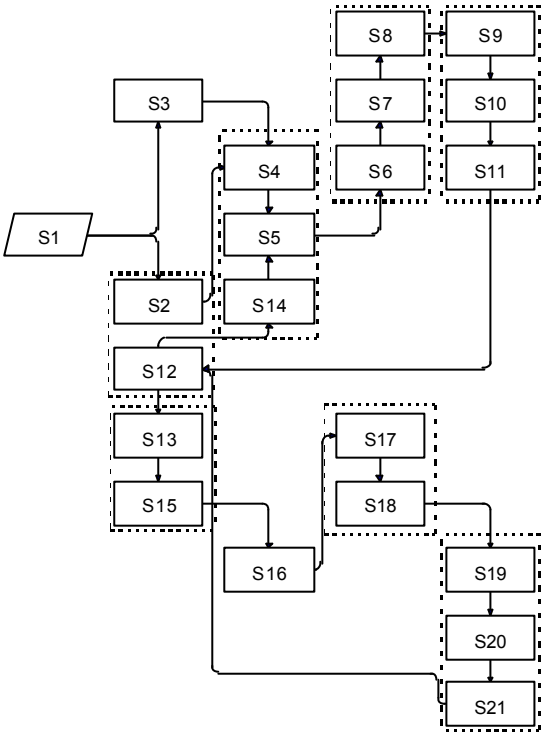
A rede apresenta estrutura igual à não-temporizada, diferindo apenas nas notações. Mais especificamente, existem agora lugares com cores temporizadas associadas (os lugares onde "circulam" os processos), os processos são agora representados por uma cor temporizada e as expressões dos arcos impõe ttls aos processos em função do tipo destes (processos-p ou processos-q).

```
// rede em [Jensen, 95: 148] simplificada conforme [Jensen, 92:
142]

code {
   const int p = 0;
   const int q = 1;
}

net AllocTim

code {
   typedef TInt U;
   typedef Int E;
}

variable<U> x;

place<U> a "a" { 3 * MultiSet<U>(U(q)) } [];
place<U> b "b" { 2 * MultiSet<U>(U(p)) } [];
```

```
place<U> c "c" {} [];
place<U> d "d" {} [];
place<U> e "e" {} [];

place<E> r "r" { MultiSet<E>(E(1)) } [];
place<E> s "s" { 3 * MultiSet<E>(E(1)) } [];
place<E> t "t" { 2 * MultiSet<E>(E(1)) } [];


transition t1 "t1" { return x.t == q; }
   in
      a (x) { return new MultiSet<U>(U(x)); };
      r     { return new MultiSet<E>(E(1)); };
      s     { return new MultiSet<E>(E(1)); };
   out
      b     { return new MultiSet<U>(U(x.t, 3)); };
   [] [] {
      cout << "t1 disparou" << endl;
   };

transition t2 "t2" {}
   in
      b (x) { return new MultiSet<U>(x); };
      s     { return new MultiSet<E>(1, 1 + !!(x.t == p)); };
   out
      c     { return new MultiSet<U>(U(x.t, 8)); };
   [] [] {
      cout << "t2 disparou" << endl;
   };

transition t3 "t3" {}
   in
      c (x) { return new MultiSet<U>(x); };
      t     { if (x == p) return new MultiSet<E>(E(1));
              else        return new MultiSet<E>(); };
   out
      r     { if (x == q) return new MultiSet<E>(E(1));
              else        return new MultiSet<E>(); };
      d     { return new MultiSet<U>(U(x.t, (x.t == p) ? 13 :
9)); };
   [] [] {
      cout << "t3 disparou" << endl;
   };

transition t4 "t4" {}
   in
      d (x) { return new MultiSet<U>(U(x)); };
      t     { return new MultiSet<E>(E(1)); };
   out
      e     { return new MultiSet<U>(U(x.t, 12)); };
   [] [] {
      cout << "t4 disparou" << endl;
   };
```

```
transition t5 "t5" {}
   in
      e (x) { return new MultiSet<U>(x); };
   out
      s     { return new MultiSet<E>(1, 2); };
      t     { return new MultiSet<E>(1, 1 + !!(x.t == p)); };
      a     { if (x == q) return new MultiSet<U>(U(q));
              else        return new MultiSet<U>();  };
      b     { if (x == p) return new MultiSet<U>(U(p));
              else        return new MultiSet<U>();  };
   [] [] {
      cout << "t5 disparou" << endl;
   };
```

Quadro 5.10 Especificação, em CpPNeTS-DL, do sistema de reserva com temporizações.



Quadro 5.11 Grafo de ocorrências do sistema de reserva, com temporizações.

A marcação correpondente aos nós agrupados por rectângulos a tracejado apenas diferem nas temporizações. Em [Jensen, 95: 157], as temporizações não são consideradas para a distinção entre marcações, pelo que o grafo apresenta um menor número de nós.

O estado S1 é representado de forma distinta dos restantes por não possuir estados anteriores. Tal resulta da utilização da aplicação allCLEAR™ já referida.

O ficheiro gerado contendo a marcação dos nós do grafo consta do apêndice E.

5.1.4 Base de dados distribuída

Este exemplo pode encontrar-se em [Jensen, 92: 21:5] e o seu grafo de ocorrências em [Jensen, 95: 18]. A rede modela uma base de dados distribuída muito simples com *n* sítios (*sites*) diferentes (na implementação aqui apresentada *n* é igual a 3). Dado que a aplicação deste exemplo consiste em demonstrar a construção de um grafo de estados de uma CpPNeTS não se apresentam detalhes relativos ao sistema modelado. Para tal recomenda-se a consulta das referências citadas.

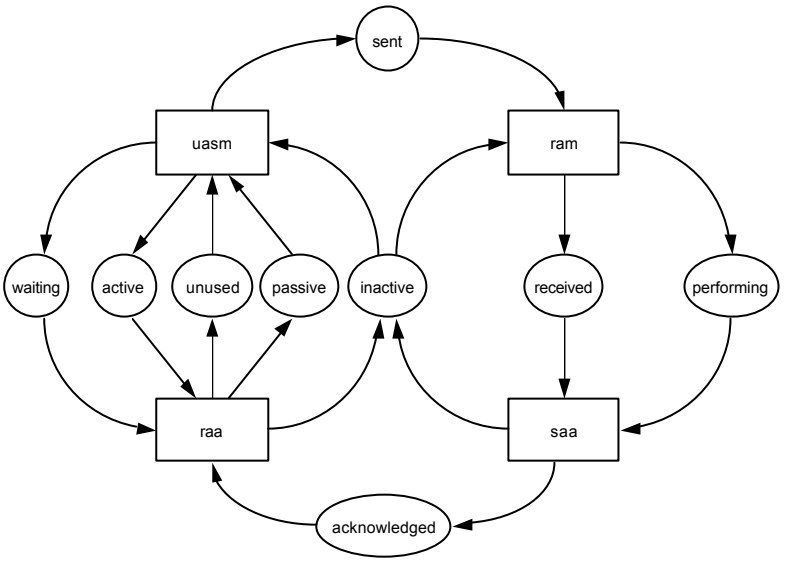


Figura 5.6 Base de dados distribuída.

```
// exemplo in [Jensen, 92: 21:5][Jensen, 95: 17:20]

net Ddb

code {
private:
    enum { n = 3 };
    typedef Int DBM;
    struct MES : public Color {
      int s;
      int r;
      MES() {}
      MES(int ss, int rr) : s(ss), r(rr) {}
      void Write(ostream& os) const { os << s << " " << r; }
      void Read(istream& is) { is >> s >> r; }
      friend bool operator==(const MES& left, const MES& right)
{
         return (left.s == right.s) && (left.r == right.r);
      }
      friend bool operator!=(const MES& left, const MES& right)
{
         return (left.s != right.s) || (left.r != right.r);
      }
   };
   typedef Int E;

   Marking       Mes(int s);
```

```
   MultiSet<MES> InitUnusedPlace();
   MultiSet<DBM> InitInactivePlace();
}

variable<DBM> s;
variable<DBM> r;
variable<MES> mes;

place<DBM> waiting "Waiting" {} [];

place<E>   active  "Active" {} [];
place<MES> unused  "Unused" { InitUnusedPlace() } [];
place<E>   passive "Passive" { MultiSet<E>(E(1)) } [];

place<DBM> inactive  "Inactive" { InitInactivePlace() } [];

place<MES> received   "Received" {} [];
place<DBM> performing "Performing" {} [];

place<MES> sent "Sent" {} [];
place<MES> acknowledged "Acknowledged" {} [];

transition uasm "Update and Send Messages" {}
   in
      inactive (s) { return new MultiSet<DBM>(s); };
      unused { return Mes(s); };
      passive { return new MultiSet<E>(E(1)); };
   out
      waiting { return new MultiSet<DBM>(s); };
      sent    { return Mes(s); };
      active  { return new MultiSet<E>(E(1)); };
   [] [] {};

transition raa "Receive all Acknowledgments" {}
   in
      waiting (s) { return new MultiSet<DBM>(s); };
      acknowledged { return Mes(s); };
      active  { return new MultiSet<E>(E(1)); };
   out
      inactive { return new MultiSet<DBM>(s); };
      unused { return Mes(s); };
      passive { return new MultiSet<E>(E(1)); };
   [] [] {};

transition ram "Receive a Message" {}
   in
      sent (mes) { return new MultiSet<MES>(mes); };
      inactive { return new MultiSet<DBM>(DBM(mes.r)); };
   out
      performing { return new MultiSet<DBM>(DBM(mes.r)); };
      received   { return new MultiSet<MES>(mes); };
   [] [] {};

transition saa "Send an Acknowledgment" {}
   in
      received   (mes) { return new MultiSet<MES>(mes); };
      performing       { return new MultiSet<DBM>(DBM(mes.r));
};
   out
      inactive         { return new MultiSet<DBM>(DBM(mes.r));
};
      acknowledged     { return new MultiSet<MES>(mes); };
   [] [] {};

code {
```

```
   Marking
   Ddb::Mes(int s) {
      MultiSet<MES> m;
      for(int i = 1; i <= n; i++)
         if (i != s) m += MultiSet<MES>(MES(s, i));
      return new MultiSet<MES>(m);
   }

   MultiSet<Ddb::MES>
   Ddb::InitUnusedPlace() {
      MultiSet<MES> m;
      for(int i = 1; i <= n; i++)
         for(int j = 1; j <= n; j++)
            if (i != j) m += MultiSet<MES>(MES(i, j));
      return m;
    }

   MultiSet<Ddb::DBM>
   Ddb::InitInactivePlace() {
       MultiSet<DBM> m;
       for(int i = 1; i <= n; i++) { m += MultiSet<DBM>(DBM(i));
}
       return m;
    }

}
```

Quadro 5.12 Especificação, em CpPNeTS-DL, da bases de dados distribuída.

Segue-se o grafo de ocorrências obtido (Figura 5.7).

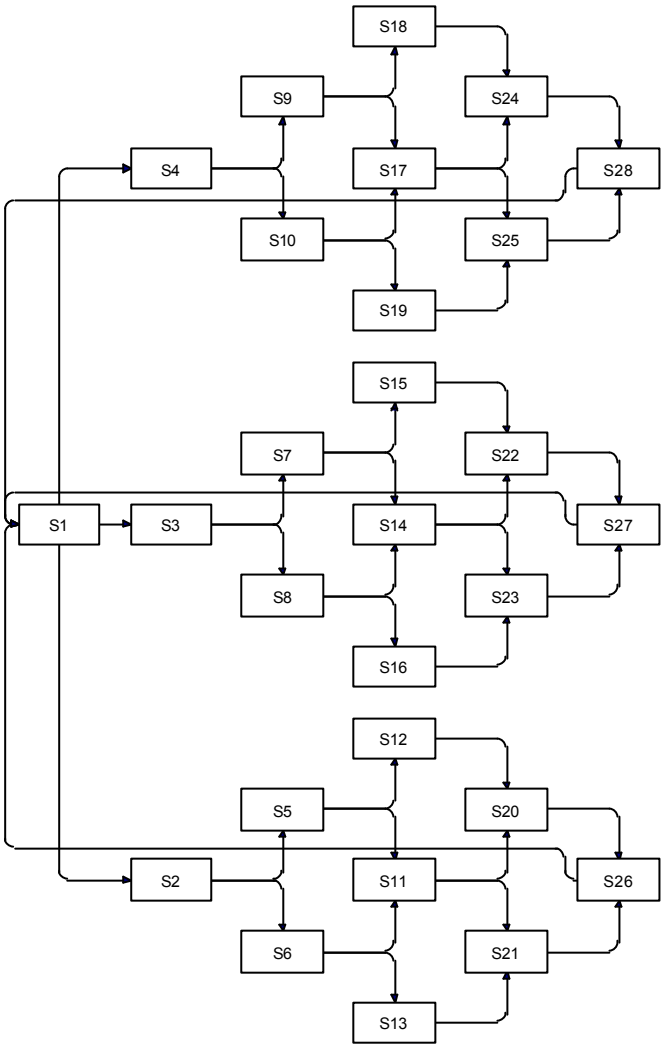


Figura 5.7 Grafo de ocorrências da base de dados distribuída para n = 3.

O ficheiro gerado contendo a marcação dos nós do grafo consta do apêndice E.

5.2 Redes sincronizadas

Apresenta-se seguidamente um exemplo de uma rede sincronizada que utiliza as capacidades de especificação de sincronização, nomeadamente através da utilização de eventos e acções.

5.2.1 Sistema de Controlo de Produção

Este exemplo de modelação pode encontrar-se em [Gomes et al., 92] [Gomes et al., 93]. Conforme descrito em [Gomes et al., 93], consiste num

> "(...)sistema de controlo de um sistema de produção constituído por centros de montagem ou maquinagem, alimentados através de tapetes transportadores. Cada centro dispõe de

sensores que permitem determinar a presença de peças ou produtos na entrada e na saída dos tapetes , bem como de quatro variáveis de entrada, C1 a C4, provenientes de um sistema de identificaçao (por exemplo, leitor de código de barras ou sistema sensorial baseado em ultra-sons ou visão). O sistema é modelado através da RdP (...) na qual se salienta a utilização de marcas com temporizações (atributo TTL), permintindo a especificação de tempos de procesamento diferentes para vários tipos de peças ou produtos, identificados através de C1 a C4."

A Figura 5.8 mostra de forma esquemática o sistema a modelar, bem como, a localização relativa dos sensores dos quais os eventos são função.

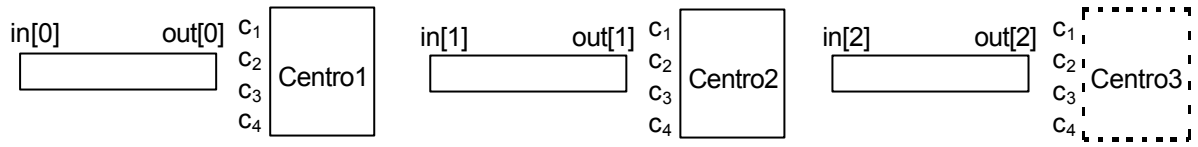


Figura 5.8 Esquema do sistema de montagem com dois centros. Assinalam-se os sensores. `in` e `out` indicam a presença de peças respectivamente, no inicio e fim tapete, `c1` `c2` `c3` e `c4` indicam o tipo de peça.

O número de centros a modelar é especificado pela constante `n`.

Relativamente à rede apresentada nas referências citadas, decompos-se a transição responsável pela introdução dos atributos `ttl` nas marcas do lugar `e` em duas. Uma das transições cria marcas com `ttl` = 5; a outra transição cria as marcas com o `ttl` = 3. Têm eventos distintos associados: uma trata das peças dos tipos C1 e C2 e a outra das peças dos tipos C3 e C4. Adicionaram-se, também, acções à maior parte dos lugares. Essas acções não se encontram especificadas nas referências originais, constituindo uma possível interpretação do modelo.

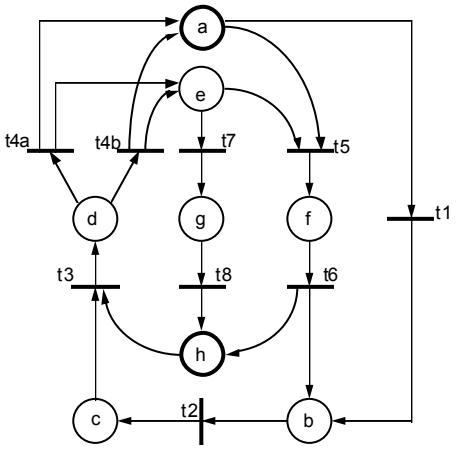


Figura 5.9 Sistema de produção.

Segue-se a especificação em CpPNeTS-DL.

```
// baseado em [Gomes et al., 93]

net Sisprod

code {

public:
static int nCentros;
static int ttl1, ttl2;

private:
typedef Int Centro;
typedef TInt TCentro;

MultiSet<Centro> Inicializar();

}

variable<Centro> i;
variable<TCentro> ti;
variable<Centro> ti.t;

event _in(int i(nCentros)); // "Inicio"
event _out(int i(nCentros)); // "Fim do tapete ocupado"
event _notOut(int i(nCentros)); // "Fim do tapete desocupado"

event _c1c2(int i(nCentros)); // peca do tipo c1 ou c2
event _c3c4(int i(nCentros)); // peca do tipo c3 ou c4

event _in_(int i(nCentros)); // "Inicio do tapete seguinte
ocupado"
event _notIn_(int ti.t(nCentros)); // "Inicio do tapete seguinte
                                   //  desocupado"
action moverTapete(int i);
action pararTapete(int i);
action alimentarComPeca(int i);
action processarPeca(int i);
action retirarPeca(int i);

place <Centro> a "Tapetes livre" { Inicializar() } [];
place <Centro> b "Tapete em movimento" {} [
   (i) { return true; } moverTapete;
];
place <Centro> c "Tapete parado e ocupado" {} [
   (i) { return true; } pararTapete;
];
place <Centro>  d "Alimentacao com peca" {} [
   (i) { return true; } alimentarComPeca;
];
place <TCentro> e "Peca em processamento" {} [
   (ti) { return true; } processarPeca;
];
place <Centro>  f "Peca totalmente processada" {} [
   (i) { return true; } retirarPeca;
];
place <Centro>  g "Peca parcialmente processada" {} [
   (i) { return true; } retirarPeca;
];
place <Centro>  h "Centros livres" { Inicializar() } [];

transition t1 "Ha peca no inicio do primeiro tapete" { return i
== 0; }
   in
```

```
      a (i) { return new MultiSet<Centro>(i); };
   out
      b { return new MultiSet<Centro>(i); };
   [_in] [] {} ;
transition t2 "Ha peca no fim do tapete i" {}
   in
      b (i) { return new MultiSet<Centro>(i); };
   out
      c { return new MultiSet<Centro>(i); };
   [_out] [] {};
transition t3 "Centro livre." {}
   in
      c (i) { return new MultiSet<Centro>(i); };
      h  { return new MultiSet<Centro>(i); };
   out
      d { return new MultiSet<Centro>(i); };
   [_out] [] {};
transition t4a "Processar pecas 1 ou 2" {}
   in
      d (i) { return new MultiSet<Centro>(i); };
   out
      a { return new MultiSet<Centro>(i); };
      e { return new MultiSet<TCentro>(TCentro(i, ttl1)); };
   [_c1c2] [] {};
transition t4b "Processar pecas 3 ou 4" {}
   in
      d (i) { return new MultiSet<Centro>(i); };
   out
      a { return new MultiSet<Centro>(i); };
      e { return new MultiSet<TCentro>(TCentro(i, ttl2)); };
   [_c3c4] [] {};
transition t5 "Terminar processamento parcial" {
   return (ti.t < (nCentros-1));
}
   in
      e (ti) { return new MultiSet<TCentro>(ti); };
      a      { return new MultiSet<Centro>(Centro(ti.t + 1)); };
   out
      f { return new MultiSet<Centro>(Centro(ti.t)); };
   [_notIn_] [] {};
transition t6 "Por peca parcialmente processada no tapete
seguinte" {
   return i < (nCentros-1);
}
   in
      f (i) { return new MultiSet<Centro>(i); };
   out
      h { return new MultiSet<Centro>(i); };
      b { return new MultiSet<Centro>(Centro(i + 1)); };
   [_in_] [] {};
transition t7 "Por peca totalmente processada no tapete
seguinte" {
   return (ti.t == (nCentros-1));
}
   in
      e (ti) { return new MultiSet<TCentro>(ti); }
   out
      g { return new MultiSet<Centro>(Centro(ti.t)); };
   [_notIn_] [] {};
transition t8 "Desocupar centro" {}
   in
      g (i) { return new MultiSet<Centro>(i); }
   out
      h { return new MultiSet<Centro>(i); }
   [_in_] [] {};
```

```
code {
   int Sisprod::nCentros;
   int Sisprod::ttl1, Sisprod::ttl2;


   MultiSet<Sisprod::Centro> Sisprod::Inicializar() {
      MultiSet<Centro> mc;
      for(int i = 0; i < nCentros; i++)
         mc += MultiSet<Centro>(Centro(i));
      return mc;
   }
}
```

Quadro 5.13 Especificação do sistema de controlo de produção em CpPNeTS-DL.

Apresenta-se seguidamente a máquina de estados obtida para um sistema com `1` centro, `ttl1` = 2 e `ttl2` = 1.

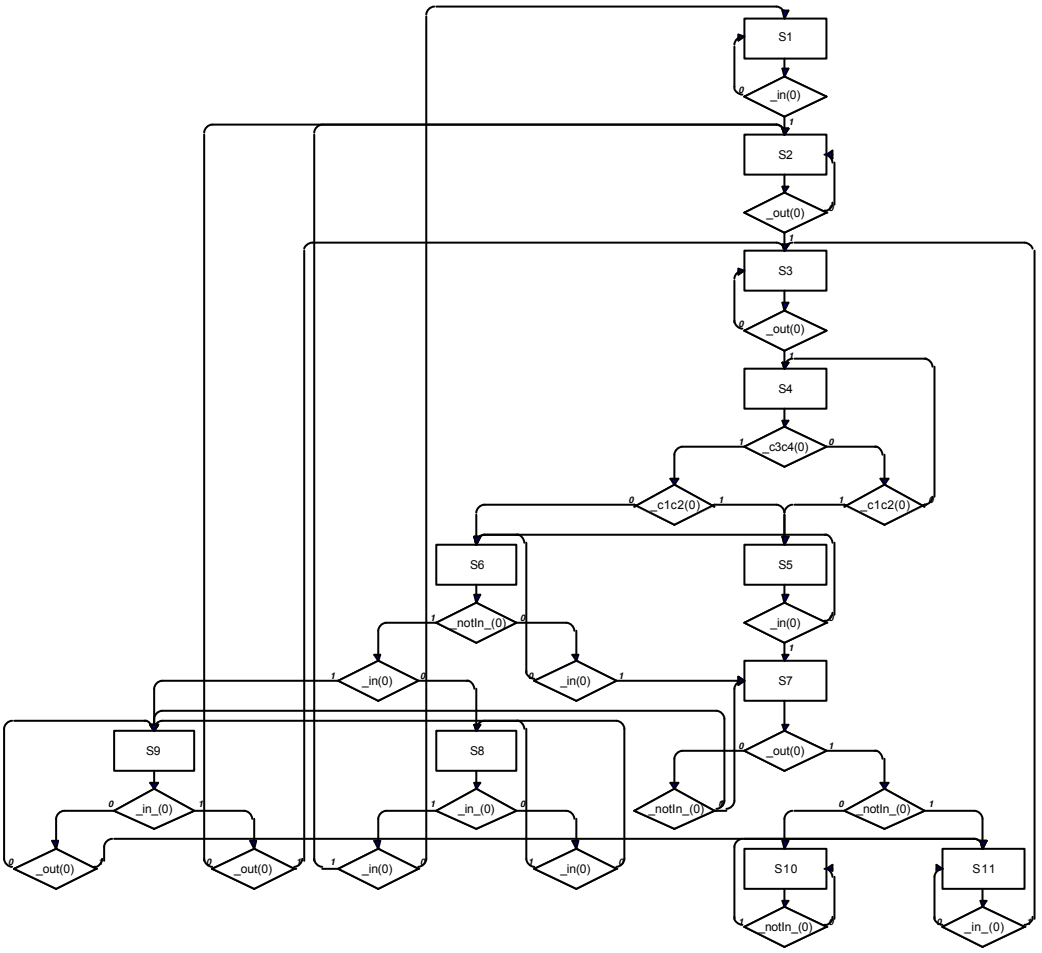


Figura 5.10 Máquina de estados do sistema de controlo de produção com 1 centro, `ttl1` = 2 e `ttl2` = 1. A figura foi obtida a partir do ficheiro `sm.acl` gerado pela bilioteca CpPNeTS-Lib, utilizando a aplicação allCLEAR™.

O conteúdo dos ficheiros gerados pelo pré-processador, para este exemplo, encontra-se no Apêndice E.

Capítulo 6

Conclusão

O sistema realizado permite já verificar a parte fundamental do sistema de suporte à geração de código para controladores programáveis: a geração da máquina de estados síncrona a partir de uma RdP de alto-nível (CpPNeTS).

Por outro lado a biblioteca desenvolvida, ao suportar a representação da estrutura de uma CpPNeTS e a determinação dos vínculos das suas contendas, oferece já um sólido suporte, para implementações de várias formas de análise das CpPNeTS, de que a, já implementada, geração do grafo de ocorrências é exemplo significativo. Também a simulação do sistema modelado através da execução da RdP é já suportada pelo sistema.

Contudo, muitas melhorias e adições ficam necessariamente por fazer. Seguidamente tece-se um breve comentário sobre os resultados obtidos e sugerem-se algumas dessas melhorias e adições ao trabalho realizado.

6.1 Resultados Obtidos

O objectivo inicial de implementar um sistema baseado em RdP de alto-nível que gerasse a máquina de estados síncrona correspondente para posterior implementação em controladores programáveis de baixo custo, foi plenamente atingido. Simultaneamente, optou-se por disponibilizar um suporte muito mais genérico para a especificação, análise e simulação da classe de RdP desenvolvida: as CpPNeTS. A utilização da linguagem C++ veio melhorar de forma significativa a capacidade de especificação da rede ao permitir a identificação das páginas e das cores com classes C++. Desta forma foi possível manter a linguagem de especificação CpPNeTS-DL muito simples e, ao mesmo

tempo, dispor de uma linguagem eficiente, versátil e largamente utilizada para a especificação das anotações da rede.

O sistema realizado veio demonstrar a viabilidade de utilização de uma RdP de alto-nível para a modelação de sistemas a eventos discretos e consequente geração da máquina de estados a implementar em *hardware* adequado. É um facto bem conhecido que o número de estados de um sistema a eventos discretos tem um crescimento exponencial à medida que aumenta a dimensão do modelo [Ho, 92b: x]. Esse problema foi verificado na prática. Uma das formas de minorar esse facto passa pela utilização de componentes exteriores ao sistema modelado e que com ele comuniquem por meio das acções e eventos externos à rede. Por exemplo, se existirem memórias externas que apenas em determinados instantes comunicam o seu estado ao sistema, este pode evitar a utilização de marcas que forneceriam a mesma informação. Essas marcas originariam um maior número de estados da máquina de estados gerada. A utilização de temporizadores externos ao sistema referida em [Gomes et al., 93], através da utilização de acções externas (para os iniciar) e de eventos externos (para os temporizadores comunicarem ao sistema o seu estado), é um exemplo de aplicação desta técnica. É interessante constatar que tal corresponde à execução paralela do controlo do sistema, por oposição a um controlo estritamente sequencial que se basearia exclusivamente na execução da máquina de estados síncrona.

Resta assinalar que uma máquina de estado demasiado extensa para poder ser implementada no *hardware* disponível, pode corresponder, de facto, ao controlo sequencial necessário. Nesse caso o próprio sistema constitui o verdadeiro problema, obrigando possivelmente a outro tipo de soluções para o seu controlo. Conforme referido em [Ho, 92b: x]:

> "Alguma forma de soluções baseadas em regras e/ou lógica difusa que tolere a imprecisão irão provavelmente constituir uma parte da caixa de ferramentas para sistemas dinâmicos a eventos discretos".

6.2 Trabalho Futuro

> *O problema do futuro é que está sempre a transformar-se no presente*
> Hobbes

A implementação e testes efectuados permitem já verificar, na prática, o funcionamento do sistema CpPNeTS-S. As funcionalidades actuais podem, não obstante, ser largamente expandidas quer

através das, sempre possíveis, optimizações do código produzido, quer introduzindo melhorias na interface com o utilizador, quer ainda adicionando novas funcionalidades. Estas podem apresentar a forma de novas funções de biblioteca que operem sobre a representação da rede, ou de novas ferramentas autónomas, designadamente aplicações para depuramento da especificação da rede. Um interpretador da máquina de estados gerada, é um exemplo desse tipo de aplicações e pode constituir uma forma de verificar a correcção do modelo.

6.2.1 Optimizações e modificações de código

A principal optimização do código produzido consistirá na modificação da estrutura de dados em disco da biblioteca (vide §4.4), de forma a possibilitar uma busca não-sequencial das marcações já existentes. Uma tabela de *hash* em memória secundária será, em principio, a melhor opção com vista a suportar um grande número de marcações mantendo, ao mesmo tempo, uma eficiência razoável.

Outra optimização possível e desejável, consiste na substituição de todas as classes auxiliares, tais como, a classe `String`, `List` e `Vector` por classes *standard* equivalentes, logo que tal seja possível. O *standard* da linguagem C++, na qual se inclui a bibioteca *stl*[82] [Meyers, 96: 280-4], não se encontra ainda terminado o que só acontecerá provavelmente em 1997 [Meyers, 96: 277]. Actualmente existem dois livros que em conjunto "definem" a linguagem, são eles: [Ellis et al., 90] e [Stroustrup, 94].

6.2.2 Interface com o utilizador

A actual interface com o utilizador é muito pouco amigável (*user-friendly*): toda a especificação da rede é feita utilizando um editor de texto. Na prática, tal pode obrigar a um desenho prévio da rede, o qual não é suportado pelo sistema, e posterior especificação textual. Desta forma, um ambiente de edição gráfica de redes integrado com o sistema será de extrema utilidade como forma de melhorar o rendimento dos utilizadores.

---

[82] Esta biblioteca inclui, designadamente, uma implementação muito eficiente de multiconjuntos, que foram denominados *maps*.

Um ambiente de desenvolvimento com edição gráfica

Conforme já referido, a biblioteca constitui a base sobre a qual se podem desenvolver as várias operações pretendidas sobre a RdP. Estas operações, bem como a especificação da CpPNeTS, serão mais eficientemente utilizadas caso se disponha de um ambiente de desenvolvimento que integre os aspectos de edição, simulação e análise das redes. Este apresentará, de preferência uma forma gráfica que torne visíveis quer a própria rede, quer a sua simulação, quer ainda os resultados das análises efectuadas. A biblioteca seria expandida de forma a servir de interface entre o editor e a representação da rede (Figura 6.1).

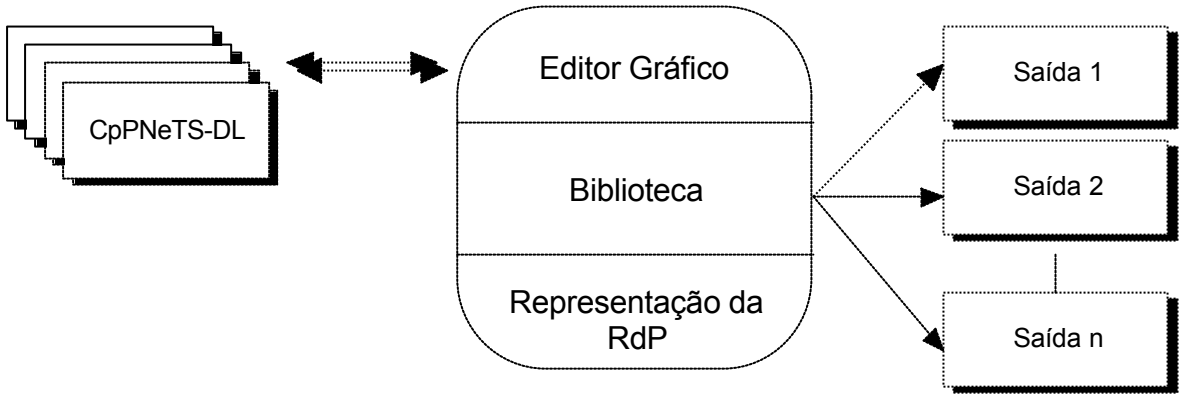


Figura 6.1 Um ambiente com editor gráfico. Neste caso a biblioteca suportaria uma especificação incremental da RdP especificada através do editor. Este escreve e lê especificações de páginas na linguagem CpPNeTS-DL. As saídas à direita da figura representam possíveis resultados de funções de análise ou simulação, implementadas na biblioteca.

Um ambiente de desenvolvimento baseado em CpPNeTS deverá oferecer, pelo menos, as seguintes funcionalidades:

- Edição gráfica e interactiva da rede, oferecendo as funcionalidades de um editor de grafos acrescidas da faculdade de especificação das inscrições da rede: funções dos arcos, marcações dos lugares, definição das cores, etc..
- Geração automática da especificação da rede na linguagem CpPNeTS-DL.
- Possibilidade de visualização dinâmica dos disparos da rede.
- Integração das diversas ferramentas constituintes do sistema.

Poder-se-ão também considerar outras capacidades mais sofisticadas tais como:

- Leitura de uma especificação na linguagem CpPNeTS-DL e consequente desenho da rede numa forma de fácil visualização, podendo até optar entre vários estilos de desenho com

vista a possibilitar ao utilizador a escolha do mais legível.

Associação a uma base de dados contendo informação específica respeitante aos dispositivos, designadamente PLCs, para os quais se pretenda gerar código. Esta ligação permitiria, após a completa especificação do sistema, a geração do código para um dispositivo específico através de um único comando por parte do utilizador.

6.2.3 Novas funcionalidades

São muitas e variadas as funcionalidades que se podem adicionar ao sistema CpPNeTS-S. Ao nível das próprias redes CpPNeTS será extremamente interessante permitir a utilização de páginas como cores dos lugares. Dessa forma as marcas dos lugares podem elas próprias ser CpPNeTS o que permitirá a modelação directa de sistemas como os dos filósofos russos [Lakos et al., 94], em que cada filósofo se encontra a pensar num problema dos filósofos e vai comer quando esse problema bloqueia (*deadlock*). Esta evolução do sistema é idêntica à verificada no sistema LOOPN [Lakos, 92a][Lakos, 92b] que serviu de base a um novo sistema denominado LOOPN++ [Lakos et al., 94] e corresponde à evolução da rede para uma rede de Petri baseada em objectos. A partir desta, e devido ao forte suporte oferecido pela linguagem C++, será relativamente simples obter uma Object Petri Net conforme definida em [Lakos, 96].

É também possível adicionar ao sistema quaisquer das técnicas de análise conhecidas para as RdP coloridas [Jensen, 92: 194-200] [Jensen, 95] bem como as variadas formas de simulação do sistema modelado através do jogo-de-marcas (*token-player*). Entre elas podemos citar, como exemplo, a simulação automática, semi-automática e manual, descritas em [Jensen, 92: 176-94].

No respeitante à implementação da máquina de estados gerada, falta a construção de um compilador que dadas as características do hardware e a máquina de estados, gere o código respectivo. Uma ferramenta intermédia, interessante para o teste da máquina de estados gerada, será um interpretador que permita a execução *off-line* da máquina de estados. Seguidamente descreve-se muito resumidamente um programa desse tipo.

Um interpretador para a máquina de estados

Como já anteriormente referido §3.2.2, a principal aplicação da máquina de estados é a geração de

código para PLCs. Neste caso o PLC funciona como interpretador correspondendo as variáveis de entrada e de saída do sistema a variáveis de entrada e de saída do PLC. Torna-se nesse caso, necessário gerar o código para o PLC em causa, carregá-lo e executá-lo no PLC. Como a geração de código para PLCs se encontra fora do âmbito deste trabalho não serão aqui referenciadas as técnicas de compilação correspondentes.

Existe, no entanto, outra forma de visualizar o sistema a "funcionar", ou seja, de visualizar a evolução das suas variáveis de saída ao longo do tempo sem a necessidade de gerar código para outra máquina: implementar um interpretador utilizando, por exemplo, a linguagem C ou a linguagem C++.

Importa notar que independentemente das várias formas de execução *hardware* que se possam vir a realizar (execução em variados PLCs ou computadores), pode muitas vezes ser importante efectuar uma primeira simulação em *software* que permita especificar uma determinada sequência de variáveis de entrada, de forma a verificar o comportamento do sistema. Desta forma é possível testar a especificação do sistema *off-line*, previamente à sua execução *on-line*.

No cado deste interpretador em *software* torna-se necessária conhecer os valores das variáveis de entrada de outra forma que não a descrita para os PLCs. Na verdade, poder-se-ia ter implementado uma solução semelhante na qual os valores das variáveis de entrada seriam conhecidos do interpretador através de uma ligação *hardware* (e.g. ligação série RS-232 ou paralela) do computador aos dispositivos físicos (e.g. sensores) responsáveis pelos sinais de entrada. Para as variáveis de saída poder-se-ia seguir uma política semelhante. Dessa forma o computador exerceria uma função muito mais semelhante à de um PLC.

Uma possível forma consiste numa descrição textual, utilizando uma linguagem própria muito simples, que permita uma especificação fácil, rápida e com um elevado potencial descritivo. A evolução das variáveis de entrada terá em conta o tempo e o período do relógio de sincronismo.

Apêndice A

Alguns tipos de Redes de Petri

Há, efectivamente, uma diferença otável entre o nome da coisa e o que deveras se passa com ela.

Richard P. Feynman

Existe já um número considerável de tipos de Redes de Petri. Por esta razão considerou-se interessante efectuar um seu levantamento que apesar de necessariamente incompleto, permitisse uma sistematização dos tipos de Redes de Petri, e respectiva nomenclatura, que mais frequentemente surgem na literatura. O glossário que em seguinda se apresenta, fornece uma breve descrição de vários tipos de rede indicando sempre uma ou mais referências onde se pode encontrar a definição apresentada (e não só).

Autónoma [David et al., 92: 4]: quando descreve o funcionamento de um sistema que evolui de uma forma autónoma, i.e., cujos instantes de disparo ou não são conhecidos ou não são especificados. Esta designação não é realmente necessária mas permite evidenciar a distinção de uma RdP com estas características, de um RdP **não-autónoma**.

Binária [David et al., 92: 21]: O m. q. RdP **segura**.

Capacidade finita [Murata, 89: 543]: a cada lugar está associada uma "capacidade", correspondente à quantidade máxima de marcas que este possa conter. Neste tipo de redes a regra de disparo das transições contem uma condição adicional: o número de marcas nos lugares de saída não pode exceder as capacidades dos respectivos lugares. À regra de disparo com esta condição é dado o nome de *regra forte de transição*; sem esta condição denomina-se *regra fraca de transição*. Demonstra-se facilmente [Murata, 89: 543] que qualquer rede de Petri **pura** e de capacidade finita pode ser transformada numa rede de Petri equivalente onde se pode aplicar a regra fraca de transição.

Capacidade infinita [Murata, 89: 543]: cada lugar pode conter uma quantidade ilimitada de marcas.

Colorida [Murata, 89: 572]: um dos três tipos de redes de alto-nível, vide § 1.7.1.

Com arco-inibidor [Peterson, 77:247][David et al., 92: 15].: RdP contendo "arcos de inibição". Estes arcos apresentam uma semântica oposta à dos arcos "normais". Os arcos de inibição permitem testar a ausência de marcas nos lugares e tornam o poder de modelação das RdP igual ao das máquinas de Turing [Peterson, 77: 247].

Com marcas individuais [Murata, 89: 572]: um dos três tipos de redes de alto-nível, vide §1.7.1.

Com prioridades [David et al., 92: 17]: as transições têm prioridades de disparo, umas sobre as outras. Tal significa que em caso de conflito entre duas ou mais transições, dis parará a que tiver uma prioridade mais elevada. Este tipo de redes não pode ser transformado numa RdP **ordinária**, dado o seu poder de modelação ser superior e idêntico ao das máquinas de Turing. Podem ser modeladas, por exemplo, por RdP **estendidas.**

Com teste-zero [David et al., 92]: O m. q. RdP **com arco-inibidor**.

Condição-Evento [Reisig, 92: 11-22]: RdP ordinárias, em que cada lugar contem no máximo uma marca. São, portanto, RdP **seguras**.

Conservativa*:* a quantidade de marcas na rede, é constante. Considerando RdP de baixo-nível, isto apenas pode suceder se o número de arcos de entrada de cada transição for igual ao número de arcos de saída. Nalguns sistemas pode ser conveniente considerar que uma única marca pode representar mais do que um recurso. A conservatividade da rede pode então medir-se com base no número de recursos. Daqui resulta outra possível definição de RdP conservativa [Zurawsky et al., 94: 572]. Se existe um vector $w = [w_1, w_2, ..., w_m]$, onde m é o número de lugares e $w(p) > 0$ para cada lugar $p \in P$, tal que a soma pesada das marcas se mantem em todas as marcações alcançadas partindo de uma marcação inicial M_0.

Contínuas [David et al., 92: 125-79]: A marcação de cada lugar não é um número inteiro positivo, mas sim um número real positivo. O disparo de uma transição corresponde à subtracção e adição de uma quantidade real de marcas aos lugares de entrada e de saída respectivamente. Estas redes permitem a modelação de sistemas que não podem ser modelados pelas RdP ordinárias.

Escolha-assimétrica [Murata, 89: 554][David et al., 92: 8-9]: se quaisquer dois lugares têm transições de saída comuns, então o conjunto de incidência posterior de pelo menos um deles deverá estar contido no conjunto de incidência posterior do outro: $\forall l_1, l_2 \in L : l_1^\bullet \cap l_2^\bullet \neq \varnothing \Rightarrow l_1^\bullet \subseteq l_2^\bullet \vee l_1^\bullet \supseteq l_2^\bullet$ [Murata, 89: 554]. O m. q. RdP **simples** [David et al., 92: 8-9].

Escolha-livre [Peterson, 77: 248][Murata, 89: 554]: cada arco é o único arco de saída de um lugar ou o único arco de entrada de uma transição: $\forall l_1, l_2 \in L: l_1^\bullet \cap l_2^\bullet \neq \varnothing \Rightarrow |l_1^\bullet| = |l_2^\bullet| = 1$. Esta restrição significa que se existe uma marca num lugar então ela continuará nesse lugar até a sua única transição de saída disparar ou, se existirem múltiplas transições de saída para o lugar, será possível escolher livremente qual das transições de saída irá disparar. Subconjunto das RdP de **escolha-livre-estendida**.

Escolha-livre-estendida: todos os lugares que partilham transições de saída têm de possuir as mesmas transições de saída: $\forall l_1, l_2 \in L: l_1^\bullet \cap l_2^\bullet \neq \varnothing \Rightarrow l_1^\bullet = l_2^\bullet$ [Murata, 89: 554]. Subconjunto das RdP de **escolha-assimétrica**.

Estocástica [Zurawsky et al., 94: 578][Marsan et al., 84][Murata, 89: 570]: redes temporizadas em que o tempo associado aos lugares ou transições é modelado através de variáveis aleatórias. Nestes modelos tornou-se convenção associar as temporizações apenas às transições [Zurawsky et al., 94: 578]. Nas RdP estocásticas, as variáveis aleatórias têm uma distribuição exponencial.

Estendidas [Peterson, 77: 246] [Murata, 89: 546]: segundo [Peterson, 77: 246], *estendidas* é uma designação que engloba as RdP que apresentam um poder de modelação superior ao das RdP ordinárias. São exemplos deste tipo de redes, as RdP **generalizadas** e as RdP com **arco-inibidor**. Segundo [Murata, 89: 546] a designação RdP estendidas resume-se a uma designação para RdP **com arcos-inibidores**[83].

Generalizada [Peterson, 77: 246][David et al., 92: 11]: é permitida a existência de mais do que um arco entre um lugar e uma transição ou entre uma transição e um lugar. Este tipo de rede tem o mesmo poder de modelação de uma RdP **ordinária**. Na sua representação gráfica é usual manter o desenho de apenas um arco entre os nós em causa, adicionando a inscrição de um número que indica a quantidade de arcos (ou peso do arco) que se pretende representar, entre os nós. O m. q. RdP **lugar-transição**.

Grafo de estados [David et al., 92: 6]: o m. q. **máquina de estados**.

Grafo de eventos [David, 91: 146][David et al., 92: 6] o m. q. **grafo marcado** ou **grafo de transições.**

Grafo de transições [David, 91: 146][David et al., 92] o m. q. **grafo marcado** ou **grafo de eventos**.

Grafo marcado [Peterson, 77: 247] [David et al., 92: 6] cada lugar tem exactamente um arco de entrada e um arco de saída. Um grafo marcado permite a representação de actividades paralelas, mas não permite representar decisões (conflitos) [Peterson, 77: 247]. Um grafo marcado é o dual do **grafo de estados** [David et al., 92: 6].

Híbridas [David et al., 92: 125-79]: contêm lugares e transições continuos (como nas RdP **contínuas**) e discretos.

Interpretada [Peterson, 77 : 230][Peterson, 81: 58-9][David, 91: 143][David et al., 92: 99]: rede à qual se associou uma interpretação, ou significado, aos seus lugares e transições, passando por isso a corresponder a algo real que se pretende modelar. Por oposição uma rede não interpretada não apresenta qualquer significado para os seus lugares e transições, é uma representação totalmente abstracta [Peterson, 77 : 230] e [Peterson, 81: 58-9].

Outro significado totalmente distinto é o apresentado em [David, 91: 143] [David et al., 92: 99]. Aí uma rede interpretada apresenta três características: é sincronizada, temporizada-L e inclui uma parte de processamento de dados cujo estado é definido por um conjunto de variáveis. Este estado é modificado por um conjunto de operações que estão associadas aos lugares e determina o valor das condições (ou predicados) que estão associados com as transições. Numa RdP interpretada, uma transição dispara quando está apta, a condição é verdadeira e o evento ocorre. Conforme afirmado na mesma referência, este modelo está muito próximo do Grafcet [David et al., 92][Novais, 94], um modelo definido para modelar controladores lógicos.

Livre-de-conflitos [David et al., 92]: cada lugar tem, no máximo, uma transição de saída.

Lugar-Transição [Reisig, 92: 25-33][Jensen, 92: 2-8]: O m. q. RdP **generalizada**.

---

[83] Provavelmente por na mesma referência ser dado o nome de *RdP* às *RdP **generalizadas*** de [Petersom, 77: 246].

k-limitada ou limitada: RdP na qual, para qualquer marcação alcançável, nenhum lugar apresenta mais de *k* marcas.

Máquina de estado [Peterson, 77: 247]: cada transição tem exactamente um arco de entrada e um arco de saída. Uma máquina de estado permite a representação de decisões (conflitos) mas não permite representar a sincronização de actividades paralelas.

Não-autónoma [David et al., 92: 17, 71-124].: uma rede que permite especificar não apenas *o que* "acontece" mas também *quando* "acontece". São redes sincronizadas com eventos externos (RdP sincronizadas) e/ou cujos passos estão dependentes do tempo (RdP temporizadas).

Ordinária [Murata, 89: 543]: Todos os arcos têm peso um. Em [Peterson, 77: 235] corresponde à definição de Rede de Petri (sem qualquer qualificação, portanto).

Predicado-Transição [Genrich, 86][Murata, 89: 572]: um dos três tipos de redes de alto-nível, vide §1.7.1.

Pura [Murata, 89: 543].: rede que não contem auto-ciclos.

Segura [David et al., 92: 21]: Característica de uma RdP em que em todas as marcações alcançáveis (a partir de uma dada marcação) nenhum lugar contem mais do que uma marca. O m. q. RdP **binária**.

Simples [David et al., 92] [Murata, 89].: cada transição é afectada, no máximo, por um conflito. O m. q. RdP de **escolha-assimétrica.**

Sincronizada [David et al., 92:]: numa rede autónoma, uma transição pode disparar se está habilitada. Numa rede sincronizada, cada transição tem um evento associado e disparará, obrigatoriamente, se estiver habilitada quando o evento ocorrer.

Temporizada [Kumar et al., 94: 1500]: permite a descrição de um sistema cujo funcionamento está dependente do tempo. Existem duas formas principais de modelar o tempo numa RdP temporizada: ou este está associado aos lugares (RdP temporizada-L) ou está associado às transições (RdP temporizada-T).

Temporizada-L: vide **temporizada**.

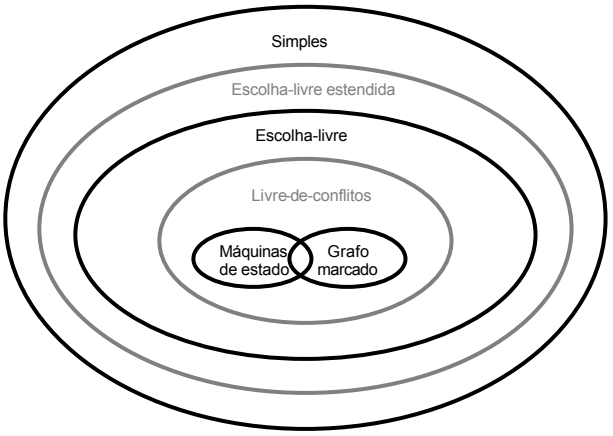
Temporizada-T: vide **temporizada**.

Persistente: se para quaisquer duas transições aptas, o disparo de qualquer delas nao incapacita a outra. Todos os grafos marcados são persistentes mas o contrário nao é verdade.

Reversivel [Zurawsky et al., 94: 573]: rede em que partindo de qualquer marcação é posível regressar à marcação inicial.

Viva: (ou, de forma equivalente, se uma dada marcação é viva) se, em qualquer marcação alcançada, é possível disparar pelo menos uma transição.

Relação de inclusão entre alguns tipos de RdP:



Apêndice B

Glossário bilingue

Permitam-me interromper aqui a minha pequena história, pois gostava de lhes falar um pouco acerca das palavras e das definições, porque, de facto é necessário aprender os nomes das coisas.

Richard P. Feynman

afectar	assign
afectado	assigned
alcançabilidade	reachability
alcançável (diz-se de uma marcação)	reachable (marking)
apta (diz-se de um transição)	enabled transition
evento	event
arco	arc
arco inibidor	inhibitor arc
árvore de alcançabilidade	reachability tree
cobertura	coverability
conflito	conflict
conflito estrutural	structural conflict
confusão	confusion
conjunto de lugares de entrada	pre-set of t
conjunto de lugares de saída	post-set of t
conjunto de transições de entrada	pre-set of p
conjunto de transições de saída	post-set of p
conservabilidade	conservativeness
consistência	consistency
controlabilidade	controllability
disparar (uma transição habilitada)	fire (of an enabled transition)
disparo (de uma transição habilitada)	firing (of an enabled transition)

disponível (diz-se de uma transição ou marca)	ready
distância sincrónica	synchronic distance
grafo de ocorrências	ocurrence graph
grafo de estados	state graph
grafo marcado	marked graph
guarda	guard
habilitada (diz-se de um transição)	enabled transition
justiça	fairness
limitabilidade	boundedness
limitada (RdP)	bounded (PN)
lugar	place
lugar complementar	complementary place
lugar fonte	source place
lugar *k*-limitado	*k*-bounded place
lugar sumidouro	sink place
macrolugar	macroplace
macrotransição	macrotransition
máquina de estado	state-machine
marca	token
marcação	marking
marcação inicial	initial marking
página	page
passo	step
persistência	persistence
pronta (diz-se de uma transição ou marca)	ready
RdP autónoma	autonomous PN
RdP colorida	coloured PN
RdP com marcas individuais	PN with individual tokens
RdP com temporizações	time PN
RdP de escolha-livre	free-choice PN
RdP de escolha-livre estendida	extended free-choice PN
RdP de escolha-assimétrica	asymmetric choice PN, simple PN
RdP generalizada	generalized PN
RdP interpretada	interpreted PN
RdP limitada	bounded PN
RdP marcada	marked PN
RdP *k*-limitada	*k*-bounded PN
RdP ordinária	PN
RdP segura	safe PN

RdP simples	simple PN, asymmetric choice PN
RdP sincronizada	synchronized PN
RdP temporizada	timed PN
RdP viva	live PN
repetibilidade	repetitiveness
reversabilidade	reversibility
sequência de disparos	firing sequence
transição	transition
transição habilitada	enabled transition
transição fonte	source transition
transição viva	live transition
vínculo	binding
vivacidade	liveness

Apêndice C

Gramática da linguagem CpPNeTS-DL

A linguagem CpPNeTS-DL é aqui definida, utilizando a metalinguagem *Backus-Naur form* ou *BNF* [Sethi, 90].

Todos os símbolos terminais se encontram assinalados a negrito. A linguagem CpPNeTS-DL tem doze palavras reservadas, são elas: **`code`**, **`net`**, **`variable`**, **`event`**, **`action`**, **`macro`**, **`place`**, **`transition`**, **`in`**, **`out`**, **`bypass_ttl`** e **`priority`**.

Os símbolos não-terminais `<int>` `<id>`, `<text>`, `<code_text>`, `<string_text>` e `<type_text>` não se encontram definidos, com vista a poupar algum espaço. Na implementação efectuada todos estes símbolos se apresentam como terminais do ponto de vista da gramática, dado serem devolvidos pelo analisador lexical já devidamente processados. O primeiro corresponde a um número inteiro, o segundo a um identificador tal como aceite pela linguagem C++ e o terceiro representa qualquer texto.

Os símbolos `<code_text>`, `<string_text>` e `<type_text>` correspondem a porções de texto livre entre chavetas no primeiro caso, aspas no segundo caso e os sinais de menor e maior, no terceiro caso. O analisador lexical resolve as condições de terminação destas porções de texto mudando o seu estado [Levine et al., 92] e contabilizando a quantidade dos caracteres parênteses em causa.

A linguagem suporta as duas formas de especificação de comentários existentes na linguagem C++: `//` e `/* */`. Os comentários são resolvidos pelo analisador lexical *flex*. Por esta razão, não surgem na gramática, dado que o texto afectado é simplesmente ignorado.

O símbolo não-terminal <empty> é utilizado apenas para tornar a gramática mais legível e traduz a ausência de simbolos.

Segue-se a gramática em BNF:

```
<pag> → <code> net <id> <code> <elements> <code>

<code> → code <code_text>
       | <empty>

<elements> → <elements> <element>
           | <element>

<element> → <macro_>
          | <variable_>
          | <action_>
          | <event_>
          | <place_>
          | <transition_>
          | <port_>

<macro_> → macro <transition_place> <id> <list_objs> ;

<list_objs> → <list_objs> , <obj>
            | <obj>

<obj> → <id> ( <list_interfaces> )

<list_interfaces> → <list_interfaces> , <interface>
                  | <interface>

<interface> → place <id> = <id>
            | <empty>

<variable_> → variable <type_text> <id> ;

<event_> → event <id> ( <event_parameters> ) ;

<event_parameters> → <event_parameters> , <event_parameter>
                   | event_parameter

<event_parameter> → <id> <id> <instances>
                  | <empty>

<instances> → ( <int> )
            | ( <id>  )
            | <empty>

<action_> → action <id> ( <action_parameters> ) ;

<action_parameters> → <action_parameters> , <action_parameter>
                    | action_parameter

<action_parameter> → <id> <id>
                   | <empty>

<port_> → port place <type_text> <id> ;

<place_> → place <type_text> <id> <string_text> <code_text>

<place_actions> → <place_actions> ; <place_action>
                | place_action
```

```
<place_action> → <vars_place_guard> <code_text> <id>
               | <empty>

<vars_place_guard> → ( <vars_pg> )
                   | <empty>

<vars_pg> → <vars_pg> , <var_pg>
          | var_pg

<var_pg> → <id>

<transition_> → transition <id> <string_text> <code_text>
                in <arcs> out <arcs>
                [ <event_name> ] [ <transition_actions> ]
<code_text>
                <priority_> ;

<arcs> → <arcs> ; <arc>
       | <arc>

<arc> → <id> <bindables> <code_text>
      | <empty>

<bindables> → ( <vars> )
            | <empty>

<vars> → <vars> , <var>
       | <var>

<var> → <op> <id>

<op> → bypass_ttl
     | <empty>

<event_name> → <id>
             | <empty>

<transition_actions> → <transition_actions> ;
<transition_action>
                     |  <transition_action>

<transition_action> → <id>
                    | <empty>

<priority_> → priority = <int>
            | <empty>

<empty> →
```

Apêndice D

Algumas notas sobre o código C++ produzido

O código foi desenvolvido utilizando a língua inglesa.

A classe *Vector*

Existem situações em que se torna desejável a eficiência de acesso de um vector e a flexibilidade e eficiência de crescimento de uma lista ligada. Por outras palavras: um vector dinâmico. A classe *Block* [Murray, 93] é uma resposta (necessariamente com alguns compromissos) a este tipo de necessidade. Consiste basicamente numa classe vector com a capacidade de crescer de forma eficiente. A eficiência de acesso é igual à de um vector simples. Apenas quando se adiciona um elemento pode esta operação ser mais lenta do que a correspondente para uma lista ligada (como a classe *List*). Para minorar esta penalização, utiliza uma estratégia de crescimento geométrico que à custa de algum desperdício de memória, reduz em muito a possibilidade de um crescimento real da estrutura que a penaliza face à lista ligada. Dado o desperdício de memória que este tipo de estrutura acarreta poder tornar-se significativo, ela deverá preferencialmente ser utilizada como suporte a estruturas temporárias e bastante dinâmicas. Esta classe tem no entanto uma desvantagem que inviabilizou a sua utilização: o crescimento do vector é implementado copiando os dados já existentes para uma nova posição de memória reservada já com espaço suficiente para em seguida adicionar os novos dados. O problema está no facto desta estratégia implicar uma alteração dos endereços de memória dos dados já presentes, inviabilizando a utilização de referências (ou apontadores) para essas posições de memória, anteriormente fornecidas para o exterior.

A class *Table* [Cargill, 94a] resolve o problema das referências para posições não reservadas de memória [Cargill, 93] à custa de uma relativamente pequena penalização no tempo de acesso à estrutura de dados. A estratégia consiste em nunca mover os dados já inseridos, para outra posição de memória. Isso implica algum processamento extra. Mesmo assim, a possibilidade da estrutura poder devolver referências para os seus dados compensa largamente o processamento extra a efectuar. Para mais pormenores recomenda-se a consulta de [Cargill, 93] e [Cargill, 94a].

Para estruturas estáticas ao longo da execução do programa (por exemplo os eventos na RdP), a classe *Vector* será preferível visto nunca acarretar desperdicio de memória e oferecer, em todas as circunstâncias, a eficiência de um vector simples.

A classe Forest

Floresta é o termo normalmente utilizado na literatura [Kruse, 84] [Sedgewick, 88] [Cormen et al., 90] para descrever uma estrutura de dados constituída por um conjunto de estruturas em árvore (ou árvores). É esse tipo de estrutura que esta classe oferece. As árvores apenas podem ser percorridas das folhas para a raíz pelo que para a estrutura de dados de suporte se utilizar um simples vector. Cada elemento deste vector conterá a informação e um apontador para o seu pai (com um valor pré-determinado no caso de ser raíz). Para percorrer os vários ramos das várias árvores basta colocar todas as folhas em outro vector sobre o qual se itera para obter o ínicio de cada um dos caminhos ao longo das árvores.

Na implementação realizada utilizaram-se, variáveis da classe *Vector*, que substituem com vantagem os vectores de baixo nível oferecidos pela linguagem C++.

Classes Iterador

As classes iterador [Stroustrup, 91][Coplien, 92][Lucas, 93] têm como única função permitir o percurso ao longo de uma dada estrutura de dados. Foram escritas classes deste tipo para as estruturas *Vector*, *List* e *Forest*.

Apêndice E

Exemplos de Aplicação

Este apêndice complementa o capítulo 5. Aqui são apresentados alguns ficheiros gerados pelo sistema aquando da tradução e/ou execução dos exemplos. A divisão em sub-capítulos é igual à do capítulo 5 de forma a facilitar uma leitura paralela. No entanto, e com o objectivo não aumentar desnecessariamente a extensão do texto, apenas alguns sub-capítulos contêm informação

E.1 Redes autónomas

E.1.1 Problema dos filósofos

Rede colorida

Rede de baixo nível hierárquica

```
// Code generated by pnetcpp V. 1.0, a Petri Net to C++ pre-
processor


#ifndef mesa_H
#define mesa_H

#include "cppnets.h"

// macro-transitions
#include "filosofo.h"
#include "filosofo.h"
#include "filosofo.h"
#include "filosofo.h"
#include "filosofo.h"

// code
class mesa: public Cppnets{

private:
   static unsigned nInst;
```

```
   unsigned inst;
   unsigned Inst() const { return inst; }

// code

#line 5 "mesa.pag"

public:
    void Simular(unsigned n);

private:
   void MostrarMesa(unsigned i);
public:

// macro-transitions
   filosofo f1;
   filosofo f2;
   filosofo f3;
   filosofo f4;
   filosofo f5;
// variables

// places
   Place* garfo1;
   Place* garfo2;
   Place* garfo3;
   Place* garfo4;
   Place* garfo5;

// port places

   mesa();

// places actions

// guards

// arc expressions

// code segments
};

#endif
```

Ficheiro *mesa.h* gerado pelo pré-processador a partir do código do Quadro 5.3.

```
// Code generated by pnetcpp V. 1.0, a Petri Net to C++ pre-
processor


#include "mesa.h"

unsigned mesa::nInst = 1;

mesa::mesa()
:
// macro-transitions
   f1(
          places, transitions,
          variables, pagEvents, actions),
   f2(
          places, transitions,
          variables, pagEvents, actions),
   f3(
          places, transitions,
          variables, pagEvents, actions),
   f4(
```

```
          places, transitions,
          variables, pagEvents, actions),
   f5(
          places, transitions,
          variables, pagEvents, actions),
   inst(nInst++)
{
   int fp = places.GetSize();
   int fv = variables.GetSize();
// variables

// actions

// events

// places
   places() = garfo1 = new Place(
          "garfo1", inst,
          "",
          new MultiSet<Int>(
#line 13 "mesa.pag"
 MS<Int>(Int(1)) )
          );
   places() = garfo2 = new Place(
          "garfo2", inst,
          "",
          new MultiSet<Int>(
#line 14 "mesa.pag"
 MS<Int>(Int(1)) )
          );
   places() = garfo3 = new Place(
          "garfo3", inst,
          "",
          new MultiSet<Int>(
#line 15 "mesa.pag"
 MS<Int>(Int(1)) )
          );
   places() = garfo4 = new Place(
          "garfo4", inst,
          "",
          new MultiSet<Int>(
#line 16 "mesa.pag"
 MS<Int>(Int(1)) )
          );
   places() = garfo5 = new Place(
          "garfo5", inst,
          "",
          new MultiSet<Int>(
#line 17 "mesa.pag"
 MS<Int>(Int(1)) )
          );

// arc expressions
// interfaces
   f1.garfo_direito = garfo1;
   f1.garfo_esquerdo = garfo2;
   f2.garfo_direito = garfo2;
   f2.garfo_esquerdo = garfo3;
   f3.garfo_direito = garfo3;
   f3.garfo_esquerdo = garfo4;
   f4.garfo_direito = garfo4;
   f4.garfo_esquerdo = garfo5;
   f5.garfo_direito = garfo5;
   f5.garfo_esquerdo = garfo1;
//Initialization
   Init();
}
```

```
// places actions

// guards

// arc expressions

// code segments

// code

#line 30 "mesa.pag"


void
mesa::Simular(unsigned n) {
   for(unsigned i = 0; i < n; i++) {
      MostrarMesa(i);
      if (ComputeConflictsBindingsS()) {
         RandomFireConflicts();
         UpdateAllConflicts();
      }
   }
   MostrarMesa(n);
}

void
mesa::MostrarMesa(unsigned i) {
   cout << "\nM" << i << '\n';
   cout << "comendo:  " << !f1.comendo->GetMarking()->IsEmpty()
<<
                           !f2.comendo->GetMarking()->IsEmpty()
<<
                           !f3.comendo->GetMarking()->IsEmpty()
<<
                           !f4.comendo->GetMarking()->IsEmpty()
<<
                           !f5.comendo->GetMarking()->IsEmpty()
<< endl;
   cout << "garfos:   " << !garfo1->GetMarking()->IsEmpty() <<
                          !garfo2->GetMarking()->IsEmpty() <<
                          !garfo3->GetMarking()->IsEmpty() <<
                          !garfo4->GetMarking()->IsEmpty() <<
                          !garfo5->GetMarking()->IsEmpty() <<
endl;
   cout << "pensando: " << !f1.pensando->GetMarking()->IsEmpty()
<<
                           !f2.pensando->GetMarking()->IsEmpty()
<<
                           !f3.pensando->GetMarking()->IsEmpty()
<<
                           !f4.pensando->GetMarking()->IsEmpty()
<<
                           !f5.pensando->GetMarking()->IsEmpty()
<< '\n'<< endl;
}
```

Ficheiro *mesa.c* gerado pelo pré-processador a partir do código do Quadro 5.3.

```
// Code generated by pnetcpp V. 1.0, a Petri Net to C++ pre-
processor


#ifndef filosofo_H
#define filosofo_H
```

```
#include "cppnets.h"

// macro-transitions

// code
class filosofo{

private:
   static unsigned nInst;
   unsigned inst;
   unsigned Inst() const { return inst; }

// code

#line 5 "filosofo.pag"

   static Expression Mark() {
      return new MS<Int>(Int(1));
   }
public:

// macro-transitions
// variables

// places
   Place* pensando;
   Place* comendo;

// port places
   Place* garfo_direito;
   Place* garfo_esquerdo;

   filosofo(Vector<Place*>& places,
          Vector<AbsTransition*>& transitions,
          Vector<AbsVariable*>& variables,
          Vector<PagEvent*>& events,
          Vector<Action*>& actions);

// places actions

// guards
   bool Guard_Comer() const;
   bool Guard_Pensar() const;

// arc expressions
   Expression ArcExp_garfo_direito_Comer();
   Expression ArcExp_garfo_esquerdo_Comer();
   Expression ArcExp_pensando_Comer();
   Expression ArcExp_Comer_comendo();
   Expression ArcExp_comendo_Pensar();
   Expression ArcExp_Pensar_pensando();
   Expression ArcExp_Pensar_garfo_direito();
   Expression ArcExp_Pensar_garfo_esquerdo();

// code segments
   void CodeSegment_Comer();
   void CodeSegment_Pensar();
};

#endif
```

Ficheiro *filósofo.h* gerado pelo pré-processador a partir do código do Quadro 5.4

```
// Code generated by pnetcpp V. 1.0, a Petri Net to C++ pre-
processor
```

```
#include "filosofo.h"

unsigned filosofo::nInst = 1;

filosofo::filosofo(
   Vector<Place*>& places,
   Vector<AbsTransition*>& transitions,
   Vector<AbsVariable*>& variables,
   Vector<PagEvent*>& pagEvents,
   Vector<Action*>& actions)
:
// macro-transitions
   inst(nInst++)
{
   int fp = places.GetSize();
   int fv = variables.GetSize();
// variables

// actions

// events

// places
   places() = pensando = new Place(
          "pensando", inst,
          "O filosofo pensa",
          new MultiSet<Int>(
#line 14 "filosofo.pag"
 MultiSet<Int>(Int(1)) )
          );
   places() = comendo = new Place(
          "comendo", inst,
          "O filosofo come",
          new MultiSet<Int>()
          );

// arc expressions
// ******************************
// transition Comer input arcs
   AbsInArc** inArcs_Comer = new AbsInArc*[4];
   inArcs_Comer[0] = new InArc<filosofo>(
          this,
          garfo_direito,
          &filosofo::ArcExp_garfo_direito_Comer);
   inArcs_Comer[1] = new InArc<filosofo>(
          this,
          garfo_esquerdo,
          &filosofo::ArcExp_garfo_esquerdo_Comer);
   inArcs_Comer[2] = new InArc<filosofo>(
          this,
          places[fp + 0],
          &filosofo::ArcExp_pensando_Comer);
   inArcs_Comer[3] = 0;
// transition Comer output arcs
   AbsOutArc** outArcs_Comer = new AbsOutArc*[2];
   outArcs_Comer[0] = new OutArc<filosofo>(
          this,
          places[fp + 1],
          &filosofo::ArcExp_Comer_comendo);
   outArcs_Comer[1] = 0;
   Action** actions_Comer = new Action* [1];
   actions_Comer[0] = 0;
// transition Comer
   transitions() = new Transition<filosofo>(
          "Comer", inst,
```

```
          "Retira garfos",
          inArcs_Comer, 3, outArcs_Comer,
          this,
          &filosofo::CodeSegment_Comer, // code segment
          0, // guard
          0, // event
          actions_Comer, // actions
          -1); // priority
// ******************************
// transition Pensar input arcs
   AbsInArc** inArcs_Pensar = new AbsInArc*[2];
   inArcs_Pensar[0] = new InArc<filosofo>(
          this,
          places[fp + 1],
          &filosofo::ArcExp_comendo_Pensar);
   inArcs_Pensar[1] = 0;
// transition Pensar output arcs
   AbsOutArc** outArcs_Pensar = new AbsOutArc*[4];
   outArcs_Pensar[0] = new OutArc<filosofo>(
          this,
          places[fp + 0],
          &filosofo::ArcExp_Pensar_pensando);
   outArcs_Pensar[1] = new OutArc<filosofo>(
          this,
          garfo_direito,
          &filosofo::ArcExp_Pensar_garfo_direito);
   outArcs_Pensar[2] = new OutArc<filosofo>(
          this,
          garfo_esquerdo,
          &filosofo::ArcExp_Pensar_garfo_esquerdo);
   outArcs_Pensar[3] = 0;
   Action** actions_Pensar = new Action* [1];
   actions_Pensar[0] = 0;
// transition Pensar
   transitions() = new Transition<filosofo>(
          "Pensar", inst,
          "Repoe garfos",
          inArcs_Pensar, 1, outArcs_Pensar,
          this,
          &filosofo::CodeSegment_Pensar, // code segment
          0, // guard
          0, // event
          actions_Pensar, // actions
          -1); // priority
// interfaces
}


// places actions

// guards

// arc expressions
Expression filosofo::ArcExp_garfo_direito_Comer() {
#line 19 "filosofo.pag"
 return Mark();
}
Expression filosofo::ArcExp_garfo_esquerdo_Comer() {
#line 20 "filosofo.pag"
 return Mark();
}
Expression filosofo::ArcExp_pensando_Comer() {
#line 21 "filosofo.pag"
 return Mark();
}
Expression filosofo::ArcExp_Comer_comendo() {
#line 23 "filosofo.pag"
```

```
 return Mark();
}
Expression filosofo::ArcExp_comendo_Pensar() {
#line 31 "filosofo.pag"
 return Mark();
}
Expression filosofo::ArcExp_Pensar_pensando() {
#line 33 "filosofo.pag"
 return Mark();
}
Expression filosofo::ArcExp_Pensar_garfo_direito() {
#line 34 "filosofo.pag"
 return Mark();
}
Expression filosofo::ArcExp_Pensar_garfo_esquerdo() {
#line 35 "filosofo.pag"
 return Mark();
}

// code segments
void filosofo::CodeSegment_Comer() {
#line 24 "filosofo.pag"

      cout << "O filosofo " << Inst() << " vai comer." << endl;
   }
void filosofo::CodeSegment_Pensar() {
#line 36 "filosofo.pag"

      cout << "O filosofo " << Inst() << " vai pensar." << endl;
   }

// code
```

Ficheiro *filósofo.c* gerado pelo pré-processador a partir do código do Quadro 5.4

E.1.2 O problema dos filósofos cooperantes

5.1.3 Sistema de Reserva de Recursos (Resource Allocation System)

Com temporizações

5.1.4 Base de dados distribuída

Apresenta-se aqui o código gerado pelo pré-processador para cada um dos exemplos bem como a marcação do exemplo da Base de dados distribuída que não se incluiu no capítulo respectivo devido à sua extensão.

```
{ file
automatically
generated by
CpPNeTS-Lib v1.0
}
```

```
S1
a_1: 3#1 0
b_1: 2#0 0
```

```
c_1: empty
d_1: empty
e_1: empty
r_1: 1#1
s_1: 3#1
```

```
t_1: 2#1

S2
a_1: 2#1 0
b_1: 1#1 1 + 2#0
0
c_1: empty
d_1: empty
e_1: empty
r_1: empty
s_1: 2#1
t_1: 2#1

S3
a_1: 3#1 0
b_1: 1#0 0
c_1: 1#0 1
d_1: empty
e_1: empty
r_1: 1#1
s_1: 1#1
t_1: 2#1

S4
a_1: 2#1 0
b_1: 1#1 1 + 1#0
0
c_1: 1#0 1
d_1: empty
e_1: empty
r_1: empty
s_1: empty
t_1: 2#1

S5
a_1: 2#1 0
b_1: 1#1 0 + 1#0
0
c_1: 1#0 0
d_1: empty
e_1: empty
r_1: empty
s_1: empty
t_1: 2#1

S6
a_1: 2#1 0
b_1: 1#1 0 + 1#0
0
c_1: empty
d_1: 1#0 2
e_1: empty
r_1: empty
s_1: empty
t_1: 1#1

S7
a_1: 2#1 0
b_1: 1#1 0 + 1#0
0
c_1: empty
d_1: 1#0 1
e_1: empty
r_1: empty
s_1: empty
t_1: 1#1

S8
a_1: 2#1 0
b_1: 1#1 0 + 1#0
0
c_1: empty
d_1: 1#0 0
e_1: empty
r_1: empty
s_1: empty
t_1: 1#1

S9
a_1: 2#1 0
b_1: 1#1 0 + 1#0
0
c_1: empty
d_1: empty
e_1: 1#0 2
r_1: empty
s_1: empty
t_1: empty

S10
a_1: 2#1 0
b_1: 1#1 0 + 1#0
0
c_1: empty
d_1: empty
e_1: 1#0 1
r_1: empty
s_1: empty
t_1: empty

S11
a_1: 2#1 0
b_1: 1#1 0 + 1#0
0
c_1: empty
d_1: empty
e_1: 1#0 0
r_1: empty
s_1: empty
t_1: empty

S12
a_1: 2#1 0
b_1: 1#1 0 + 2#0
0
c_1: empty
d_1: empty
e_1: empty
r_1: empty
s_1: 2#1
t_1: 2#1

S13
a_1: 2#1 0
b_1: 2#0 0
c_1: 1#1 1
d_1: empty
e_1: empty
r_1: empty
s_1: 1#1
t_1: 2#1

S14
a_1: 2#1 0
b_1: 1#1 0 + 1#0
0
c_1: 1#0 1
d_1: empty
e_1: empty
r_1: empty
s_1: empty
t_1: 2#1

S15
a_1: 2#1 0
b_1: 2#0 0
c_1: 1#1 0
d_1: empty
e_1: empty
r_1: empty
s_1: 1#1
t_1: 2#1

S16
a_1: 2#1 0
b_1: 2#0 0
c_1: empty
d_1: 1#1 1
e_1: empty
r_1: 1#1
s_1: 1#1
t_1: 2#1

S17
a_1: 1#1 0
b_1: 1#1 1 + 2#0
0
c_1: empty
d_1: 1#1 1
e_1: empty
r_1: empty
s_1: empty
t_1: 2#1

S18
a_1: 1#1 0
b_1: 1#1 0 + 2#0
0
c_1: empty
d_1: 1#1 0
e_1: empty
r_1: empty
s_1: empty
t_1: 2#1

S19
a_1: 1#1 0
b_1: 1#1 0 + 2#0
0
c_1: empty
d_1: empty
e_1: 1#1 2
r_1: empty
s_1: empty
t_1: 1#1

S20
a_1: 1#1 0
b_1: 1#1 0 + 2#0
0
c_1: empty
d_1: empty
e_1: 1#1 1
r_1: empty
s_1: empty
t_1: 1#1

S21
a_1: 1#1 0
b_1: 1#1 0 + 2#0
0
c_1: empty
d_1: empty
e_1: 1#1 0
r_1: empty
s_1: empty
t_1: 1#1
```

Marcação dos nós do grafo de ocorrências do sistema de reserva com temporizações.

```
{ file automatically generated by
CpPNeTS-Lib v1.0 }


S1
waiting_1: empty
active_1: empty
unused_1: 1#3 2 + 1#3 1 + 1#2 3 +
1#2 1 + 1#1 3 + 1#1 2
passive_1: 1#1
inactive_1: 1#3 + 1#2 + 1#1
received_1: empty
performing_1: empty
sent_1: empty
acknowledged_1: empty

S2
waiting_1: 1#3
active_1: 1#1
```

```
unused_1: 1#2 3 + 1#2 1 + 1#1 3 +
1#1 2
passive_1: empty
inactive_1: 1#2 + 1#1
received_1: empty
performing_1: empty
sent_1: 1#3 1 + 1#3 2
acknowledged_1: empty

S3
waiting_1: 1#2
active_1: 1#1
unused_1: 1#3 2 + 1#3 1 + 1#1 3 +
1#1 2
passive_1: empty
inactive_1: 1#3 + 1#1
received_1: empty
performing_1: empty
sent_1: 1#2 1 + 1#2 3
acknowledged_1: empty

S4
waiting_1: 1#1
active_1: 1#1
unused_1: 1#3 2 + 1#3 1 + 1#2 3 +
1#2 1
passive_1: empty
inactive_1: 1#3 + 1#2
received_1: empty
performing_1: empty
sent_1: 1#1 2 + 1#1 3
acknowledged_1: empty

S5
waiting_1: 1#3
active_1: 1#1
unused_1: 1#2 3 + 1#2 1 + 1#1 3 +
1#1 2
passive_1: empty
inactive_1: 1#2
received_1: 1#3 1
performing_1: 1#1
sent_1: 1#3 2
acknowledged_1: empty

S6
waiting_1: 1#3
active_1: 1#1
unused_1: 1#2 3 + 1#2 1 + 1#1 3 +
1#1 2
passive_1: empty
inactive_1: 1#1
received_1: 1#3 2
performing_1: 1#2
sent_1: 1#3 1
acknowledged_1: empty

S7
waiting_1: 1#2
active_1: 1#1
unused_1: 1#3 2 + 1#3 1 + 1#1 3 +
1#1 2
passive_1: empty
inactive_1: 1#3
received_1: 1#2 1
performing_1: 1#1
sent_1: 1#2 3
acknowledged_1: empty

S8
waiting_1: 1#2
active_1: 1#1
unused_1: 1#3 2 + 1#3 1 + 1#1 3 +
1#1 2
passive_1: empty
inactive_1: 1#1
received_1: 1#2 3
performing_1: 1#3
sent_1: 1#2 1
acknowledged_1: empty

S9
waiting_1: 1#1
active_1: 1#1
unused_1: 1#3 2 + 1#3 1 + 1#2 3 +
1#2 1
passive_1: empty
inactive_1: 1#3
received_1: 1#1 2
performing_1: 1#2
sent_1: 1#1 3
acknowledged_1: empty

S10
waiting_1: 1#1
active_1: 1#1
unused_1: 1#3 2 + 1#3 1 + 1#2 3 +
1#2 1
passive_1: empty
inactive_1: 1#2
received_1: 1#1 3
performing_1: 1#3
sent_1: 1#1 2
acknowledged_1: empty

S11
waiting_1: 1#3
active_1: 1#1
unused_1: 1#2 3 + 1#2 1 + 1#1 3 +
1#1 2
passive_1: empty
inactive_1: empty
received_1: 1#3 1 + 1#3 2
performing_1: 1#2 + 1#1
sent_1: empty
acknowledged_1: empty

S12
waiting_1: 1#3
active_1: 1#1
unused_1: 1#2 3 + 1#2 1 + 1#1 3 +
1#1 2
passive_1: empty
inactive_1: 1#2 + 1#1
received_1: empty
performing_1: empty
sent_1: 1#3 2
acknowledged_1: 1#3 1

S13
waiting_1: 1#3
active_1: 1#1
unused_1: 1#2 3 + 1#2 1 + 1#1 3 +
1#1 2
passive_1: empty
```

```
inactive_1: 1#2 + 1#1
received_1: empty
performing_1: empty
sent_1: 1#3 1
acknowledged_1: 1#3 2

S14
waiting_1: 1#2
active_1: 1#1
unused_1: 1#3 2 + 1#3 1 + 1#1 3 +
1#1 2
passive_1: empty
inactive_1: empty
received_1: 1#2 1 + 1#2 3
performing_1: 1#3 + 1#1
sent_1: empty
acknowledged_1: empty

S15
waiting_1: 1#2
active_1: 1#1
unused_1: 1#3 2 + 1#3 1 + 1#1 3 +
1#1 2
passive_1: empty
inactive_1: 1#3 + 1#1
received_1: empty
performing_1: empty
sent_1: 1#2 3
acknowledged_1: 1#2 1

S16
waiting_1: 1#2
active_1: 1#1
unused_1: 1#3 2 + 1#3 1 + 1#1 3 +
1#1 2
passive_1: empty
inactive_1: 1#3 + 1#1
received_1: empty
performing_1: empty
sent_1: 1#2 1
acknowledged_1: 1#2 3

S17
waiting_1: 1#1
active_1: 1#1
unused_1: 1#3 2 + 1#3 1 + 1#2 3 +
1#2 1
passive_1: empty
inactive_1: empty
received_1: 1#1 2 + 1#1 3
performing_1: 1#3 + 1#2
sent_1: empty
acknowledged_1: empty

S18
waiting_1: 1#1
active_1: 1#1
unused_1: 1#3 2 + 1#3 1 + 1#2 3 +
1#2 1
passive_1: empty
inactive_1: 1#3 + 1#2
received_1: empty
performing_1: empty
sent_1: 1#1 3
acknowledged_1: 1#1 2

S19
waiting_1: 1#1
active_1: 1#1
unused_1: 1#3 2 + 1#3 1 + 1#2 3 +
1#2 1
passive_1: empty
inactive_1: 1#3 + 1#2
received_1: empty
performing_1: empty
sent_1: 1#1 2
acknowledged_1: 1#1 3

S20
waiting_1: 1#3
active_1: 1#1
unused_1: 1#2 3 + 1#2 1 + 1#1 3 +
1#1 2
passive_1: empty
inactive_1: 1#1
received_1: 1#3 2
performing_1: 1#2
sent_1: empty
acknowledged_1: 1#3 1

S21
waiting_1: 1#3
active_1: 1#1
unused_1: 1#2 3 + 1#2 1 + 1#1 3 +
1#1 2
passive_1: empty
inactive_1: 1#2
received_1: 1#3 1
performing_1: 1#1
sent_1: empty
acknowledged_1: 1#3 2

S22
waiting_1: 1#2
active_1: 1#1
unused_1: 1#3 2 + 1#3 1 + 1#1 3 +
1#1 2
passive_1: empty
inactive_1: 1#1
received_1: 1#2 3
performing_1: 1#3
sent_1: empty
acknowledged_1: 1#2 1

S23
waiting_1: 1#2
active_1: 1#1
unused_1: 1#3 2 + 1#3 1 + 1#1 3 +
1#1 2
passive_1: empty
inactive_1: 1#3
received_1: 1#2 1
performing_1: 1#1
sent_1: empty
acknowledged_1: 1#2 3

S24
waiting_1: 1#1
active_1: 1#1
unused_1: 1#3 2 + 1#3 1 + 1#2 3 +
1#2 1
passive_1: empty
inactive_1: 1#2
received_1: 1#1 3
performing_1: 1#3
```

```
sent_1: empty
acknowledged_1: 1#1 2

S25
waiting_1: 1#1
active_1: 1#1
unused_1: 1#3 2 + 1#3 1 + 1#2 3 +
1#2 1
passive_1: empty
inactive_1: 1#3
received_1: 1#1 2
performing_1: 1#2
sent_1: empty
acknowledged_1: 1#1 3

S26
waiting_1: 1#3
active_1: 1#1
unused_1: 1#2 3 + 1#2 1 + 1#1 3 +
1#1 2
passive_1: empty
inactive_1: 1#2 + 1#1
received_1: empty
performing_1: empty
sent_1: empty
acknowledged_1: 1#3 1 + 1#3 2

S27
waiting_1: 1#2
active_1: 1#1
unused_1: 1#3 2 + 1#3 1 + 1#1 3 +
1#1 2
passive_1: empty
inactive_1: 1#3 + 1#1
received_1: empty
performing_1: empty
sent_1: empty
acknowledged_1: 1#2 1 + 1#2 3

S28
waiting_1: 1#1
active_1: 1#1
unused_1: 1#3 2 + 1#3 1 + 1#2 3 +
1#2 1
passive_1: empty
inactive_1: 1#3 + 1#2
received_1: empty
performing_1: empty
sent_1: empty
acknowledged_1: 1#1 2 + 1#1 3
```

Marcação dos nós do grafo de ocorrências da base de dados distribuída.

E.2 Redes sincronizadas

E.2.1 Sistema de Controlo de Produção

```
// Code generated by pnetcpp V. 1.0, a Petri Net to C++ pre-
processor


#ifndef Sisprod_H
#define Sisprod_H

#include "cppnets.h"

// macro-transitions

// code
class Sisprod: public Cppnets{

private:
   static unsigned nInst;
   unsigned inst;
   unsigned Inst() const { return inst; }

// code

#line 5 "sisprod.pag"


public:
static int nCentros;
static int ttl1, ttl2;

private:
```

```
typedef Int Centro;
typedef TInt TCentro;

MultiSet<Centro> Inicializar();

public:

// macro-transitions
// variables
   Centro i;
   TCentro ti;

// places
   Place* a;
   Place* b;
   Place* c;
   Place* d;
   Place* e;
   Place* f;
   Place* g;
   Place* h;

// port places

   Sisprod();

// places actions
   bool Action0_b_moverTapete();
   bool Action0_c_pararTapete();
   bool Action0_d_alimentarComPeca();
   bool Action0_e_processarPeca();
   bool Action0_f_retirarPeca();
   bool Action0_g_retirarPeca();

// guards
   bool Guard_t1() const;
   bool Guard_t2() const;
   bool Guard_t3() const;
   bool Guard_t4a() const;
   bool Guard_t4b() const;
   bool Guard_t5() const;
   bool Guard_t6() const;
   bool Guard_t7() const;
   bool Guard_t8() const;

// arc expressions
   Expression ArcExp_a_t1();
   Expression ArcExp_t1_b();
   Expression ArcExp_b_t2();
   Expression ArcExp_t2_c();
   Expression ArcExp_c_t3();
   Expression ArcExp_h_t3();
   Expression ArcExp_t3_d();
   Expression ArcExp_d_t4a();
   Expression ArcExp_t4a_a();
   Expression ArcExp_t4a_e();
   Expression ArcExp_d_t4b();
   Expression ArcExp_t4b_a();
   Expression ArcExp_t4b_e();
   Expression ArcExp_e_t5();
   Expression ArcExp_a_t5();
   Expression ArcExp_t5_f();
   Expression ArcExp_f_t6();
   Expression ArcExp_t6_h();
   Expression ArcExp_t6_b();
   Expression ArcExp_e_t7();
   Expression ArcExp_t7_g();
```

```
   Expression ArcExp_g_t8();
   Expression ArcExp_t8_h();

// code segments
};

#endif
```

Ficheiro *sisprod..h* gerado pelo pré-processador a partir do código do Quadro 5.13.

```
// Code generated by pnetcpp V. 1.0, a Petri Net to C++ pre-
processor


#include "Sisprod.h"

unsigned Sisprod::nInst = 1;

Sisprod::Sisprod()
:
// macro-transitions
   inst(nInst++)
{
   int fp = places.GetSize();
   int fv = variables.GetSize();
   int fe = pagEvents.GetSize();
// variables
   variables() = new Variable<Centro>(i, "Sisprod.i", inst);
   variables() = new Variable<TCentro>(ti, "Sisprod.ti", inst);
   variables() = new Variable<Centro>(ti.t, "Sisprod.ti.t",
inst);

// actions
   AbsParameter** param_moverTapete = new AbsParameter*[2];
   param_moverTapete[0] = new Parameter<Centro>("Centro",
variables[fv + 0]);
   param_moverTapete[1] = 0;
   actions() = new Action(
          "moverTapete",
          param_moverTapete);
   AbsParameter** param_pararTapete = new AbsParameter*[2];
   param_pararTapete[0] = new Parameter<Centro>("Centro",
variables[fv + 0]);
   param_pararTapete[1] = 0;
   actions() = new Action(
          "pararTapete",
          param_pararTapete);
   AbsParameter** param_alimentarComPeca = new AbsParameter*[2];
   param_alimentarComPeca[0] = new Parameter<Centro>("Centro",
variables[fv + 0]);
   param_alimentarComPeca[1] = 0;
   actions() = new Action(
          "alimentarComPeca",
          param_alimentarComPeca);
   AbsParameter** param_processarPeca = new AbsParameter*[2];
   param_processarPeca[0] = new Parameter<Centro>("Centro",
variables[fv + 0]);
   param_processarPeca[1] = 0;
   actions() = new Action(
          "processarPeca",
          param_processarPeca);
   AbsParameter** param_retirarPeca = new AbsParameter*[2];
   param_retirarPeca[0] = new Parameter<Centro>("Centro",
variables[fv + 0]);
   param_retirarPeca[1] = 0;
   actions() = new Action(
          "retirarPeca",
```

```
          param_retirarPeca);

// events
   EventParameter** param__in = new EventParameter*[2];
   param__in[0] = new EventParameter(
           nCentros, variables[fv + 0]);
   param__in[1] = 0;
   pagEvents() = new PagEvent("_in", param__in);

   EventParameter** param__out = new EventParameter*[2];
   param__out[0] = new EventParameter(
           nCentros, variables[fv + 0]);
   param__out[1] = 0;
   pagEvents() = new PagEvent("_out", param__out);

   EventParameter** param__notOut = new EventParameter*[2];
   param__notOut[0] = new EventParameter(
           nCentros, variables[fv + 0]);
   param__notOut[1] = 0;
   pagEvents() = new PagEvent("_notOut", param__notOut);

   EventParameter** param__c1c2 = new EventParameter*[2];
   param__c1c2[0] = new EventParameter(
           nCentros, variables[fv + 0]);
   param__c1c2[1] = 0;
   pagEvents() = new PagEvent("_c1c2", param__c1c2);

   EventParameter** param__c3c4 = new EventParameter*[2];
   param__c3c4[0] = new EventParameter(
           nCentros, variables[fv + 0]);
   param__c3c4[1] = 0;
   pagEvents() = new PagEvent("_c3c4", param__c3c4);

   EventParameter** param__in_ = new EventParameter*[2];
   param__in_[0] = new EventParameter(
           nCentros, variables[fv + 0]);
   param__in_[1] = 0;
   pagEvents() = new PagEvent("_in_", param__in_);

   EventParameter** param__notIn_ = new EventParameter*[2];
   param__notIn_[0] = new EventParameter(
           nCentros, variables[fv + 2]);
   param__notIn_[1] = 0;
   pagEvents() = new PagEvent("_notIn_", param__notIn_);


// places
   places() = a = new Place(
           "a", inst,
           "Tapetes livre",
           new MultiSet<Centro>(
#line 41 "sisprod.pag"
 Inicializar() )
           );
   AbsPlaceAction** place_b_actions = new AbsPlaceAction*[2];
   const AbsBindableVariable** var_place_b_action_0 = new const
AbsBindableVariable* [2];
   var_place_b_action_0[0] = new
           TotalBindableVariable(variables[fv + 0]);
   var_place_b_action_0[1] = 0;
   place_b_actions[0] = new PlaceAction<Sisprod>(
           var_place_b_action_0,
           1,
           this,
           &Sisprod::Action0_b_moverTapete,
           actions[0]);
   place_b_actions[1] = 0;
```

```
   places() = b = new Place(
          "b", inst,
          "Tapete em movimento",
          new MultiSet<Centro>(),
          place_b_actions);
   AbsPlaceAction** place_c_actions = new AbsPlaceAction*[2];
   const AbsBindableVariable** var_place_c_action_0 = new const
AbsBindableVariable* [2];
   var_place_c_action_0[0] = new
          TotalBindableVariable(variables[fv + 0]);
   var_place_c_action_0[1] = 0;
   place_c_actions[0] = new PlaceAction<Sisprod>(
          var_place_c_action_0,
          1,
          this,
          &Sisprod::Action0_c_pararTapete,
          actions[1]);
   place_c_actions[1] = 0;
   places() = c = new Place(
          "c", inst,
          "Tapete parado e ocupado",
          new MultiSet<Centro>(),
          place_c_actions);
   AbsPlaceAction** place_d_actions = new AbsPlaceAction*[2];
   const AbsBindableVariable** var_place_d_action_0 = new const
AbsBindableVariable* [2];
   var_place_d_action_0[0] = new
          TotalBindableVariable(variables[fv + 0]);
   var_place_d_action_0[1] = 0;
   place_d_actions[0] = new PlaceAction<Sisprod>(
          var_place_d_action_0,
          1,
          this,
          &Sisprod::Action0_d_alimentarComPeca,
          actions[2]);
   place_d_actions[1] = 0;
   places() = d = new Place(
          "d", inst,
          "Alimentacao com peca",
          new MultiSet<Centro>(),
          place_d_actions);
   AbsPlaceAction** place_e_actions = new AbsPlaceAction*[2];
   const AbsBindableVariable** var_place_e_action_0 = new const
AbsBindableVariable* [2];
   var_place_e_action_0[0] = new
          TotalBindableVariable(variables[fv + 1]);
   var_place_e_action_0[1] = 0;
   place_e_actions[0] = new PlaceAction<Sisprod>(
          var_place_e_action_0,
          1,
          this,
          &Sisprod::Action0_e_processarPeca,
          actions[3]);
   place_e_actions[1] = 0;
   places() = e = new Place(
          "e", inst,
          "Peca em processamento",
          new MultiSet<TCentro>(),
          place_e_actions);
   AbsPlaceAction** place_f_actions = new AbsPlaceAction*[2];
   const AbsBindableVariable** var_place_f_action_0 = new const
AbsBindableVariable* [2];
   var_place_f_action_0[0] = new
          TotalBindableVariable(variables[fv + 0]);
   var_place_f_action_0[1] = 0;
   place_f_actions[0] = new PlaceAction<Sisprod>(
          var_place_f_action_0,
```

```
          1,
          this,
          &Sisprod::Action0_f_retirarPeca,
          actions[4]);
   place_f_actions[1] = 0;
   places() = f = new Place(
          "f", inst,
          "Peca totalmente processada",
          new MultiSet<Centro>(),
          place_f_actions);
   AbsPlaceAction** place_g_actions = new AbsPlaceAction*[2];
   const AbsBindableVariable** var_place_g_action_0 = new const
AbsBindableVariable* [2];
   var_place_g_action_0[0] = new
          TotalBindableVariable(variables[fv + 0]);
   var_place_g_action_0[1] = 0;
   place_g_actions[0] = new PlaceAction<Sisprod>(
          var_place_g_action_0,
          1,
          this,
          &Sisprod::Action0_g_retirarPeca,
          actions[4]);
   place_g_actions[1] = 0;
   places() = g = new Place(
          "g", inst,
          "Peca parcialmente processada",
          new MultiSet<Centro>(),
          place_g_actions);
   places() = h = new Place(
          "h", inst,
          "Centros livres",
          new MultiSet<Centro>(
#line 62 "sisprod.pag"
 Inicializar() )
          );

// arc expressions
// ********************************
// transition t1 input arcs
   AbsInArc** inArcs_t1 = new AbsInArc*[2];
   const AbsBindableVariable** var_arc_a_t1 = new const
AbsBindableVariable* [2];
   var_arc_a_t1[0] = new ParcialBindableVariable(variables[fv +
0], 0);
   var_arc_a_t1[1] = 0;
   inArcs_t1[0] = new InArc<Sisprod>(
          this,
          places[fp + 0],
          &Sisprod::ArcExp_a_t1,
          var_arc_a_t1, 1);
   inArcs_t1[1] = 0;
// transition t1 output arcs
   AbsOutArc** outArcs_t1 = new AbsOutArc*[2];
   outArcs_t1[0] = new OutArc<Sisprod>(
          this,
          places[fp + 1],
          &Sisprod::ArcExp_t1_b);
   outArcs_t1[1] = 0;
   Action** actions_t1 = new Action* [1];
   actions_t1[0] = 0;
// transition t1
   transitions() = new Transition<Sisprod>(
          "t1", inst,
          "Ha peca no inicio do primeiro tapete",
          inArcs_t1, 1, outArcs_t1,
          this,
          0, // code segment
```

```
          &Sisprod::Guard_t1, // guard
          pagEvents[fe + 0], // event
          actions_t1, // actions
          -1); // priority
// ******************************
// transition t2 input arcs
   AbsInArc** inArcs_t2 = new AbsInArc*[2];
   const AbsBindableVariable** var_arc_b_t2 = new const
AbsBindableVariable* [2];
   var_arc_b_t2[0] = new ParcialBindableVariable(variables[fv +
0], 0);
   var_arc_b_t2[1] = 0;
   inArcs_t2[0] = new InArc<Sisprod>(
          this,
          places[fp + 1],
          &Sisprod::ArcExp_b_t2,
          var_arc_b_t2, 1);
   inArcs_t2[1] = 0;
// transition t2 output arcs
   AbsOutArc** outArcs_t2 = new AbsOutArc*[2];
   outArcs_t2[0] = new OutArc<Sisprod>(
          this,
          places[fp + 2],
          &Sisprod::ArcExp_t2_c);
   outArcs_t2[1] = 0;
   Action** actions_t2 = new Action* [1];
   actions_t2[0] = 0;
// transition t2
   transitions() = new Transition<Sisprod>(
          "t2", inst,
          "Ha peca no fim do tapete i",
          inArcs_t2, 1, outArcs_t2,
          this,
          0, // code segment
          0, // guard
          pagEvents[fe + 1], // event
          actions_t2, // actions
          -1); // priority
// ******************************
// transition t3 input arcs
   AbsInArc** inArcs_t3 = new AbsInArc*[3];
   const AbsBindableVariable** var_arc_c_t3 = new const
AbsBindableVariable* [2];
   var_arc_c_t3[0] = new ParcialBindableVariable(variables[fv +
0], 0);
   var_arc_c_t3[1] = 0;
   inArcs_t3[0] = new InArc<Sisprod>(
          this,
          places[fp + 2],
          &Sisprod::ArcExp_c_t3,
          var_arc_c_t3, 1);
   inArcs_t3[1] = new InArc<Sisprod>(
          this,
          places[fp + 7],
          &Sisprod::ArcExp_h_t3);
   inArcs_t3[2] = 0;
// transition t3 output arcs
   AbsOutArc** outArcs_t3 = new AbsOutArc*[2];
   outArcs_t3[0] = new OutArc<Sisprod>(
          this,
          places[fp + 3],
          &Sisprod::ArcExp_t3_d);
   outArcs_t3[1] = 0;
   Action** actions_t3 = new Action* [1];
   actions_t3[0] = 0;
// transition t3
   transitions() = new Transition<Sisprod>(
```

```
          "t3", inst,
          "Centro livre.",
          inArcs_t3, 2, outArcs_t3,
          this,
          0, // code segment
          0, // guard
          pagEvents[fe + 1], // event
          actions_t3, // actions
          -1); // priority
// ********************************
// transition t4a input arcs
   AbsInArc** inArcs_t4a = new AbsInArc*[2];
   const AbsBindableVariable** var_arc_d_t4a = new const
AbsBindableVariable* [2];
   var_arc_d_t4a[0] = new ParcialBindableVariable(variables[fv +
0], 0);
   var_arc_d_t4a[1] = 0;
   inArcs_t4a[0] = new InArc<Sisprod>(
          this,
          places[fp + 3],
          &Sisprod::ArcExp_d_t4a,
          var_arc_d_t4a, 1);
   inArcs_t4a[1] = 0;
// transition t4a output arcs
   AbsOutArc** outArcs_t4a = new AbsOutArc*[3];
   outArcs_t4a[0] = new OutArc<Sisprod>(
          this,
          places[fp + 0],
          &Sisprod::ArcExp_t4a_a);
   outArcs_t4a[1] = new OutArc<Sisprod>(
          this,
          places[fp + 4],
          &Sisprod::ArcExp_t4a_e);
   outArcs_t4a[2] = 0;
   Action** actions_t4a = new Action* [1];
   actions_t4a[0] = 0;
// transition t4a
   transitions() = new Transition<Sisprod>(
          "t4a", inst,
          "Processar pecas 1 ou 2",
          inArcs_t4a, 1, outArcs_t4a,
          this,
          0, // code segment
          0, // guard
          pagEvents[fe + 3], // event
          actions_t4a, // actions
          -1); // priority
// ********************************
// transition t4b input arcs
   AbsInArc** inArcs_t4b = new AbsInArc*[2];
   const AbsBindableVariable** var_arc_d_t4b = new const
AbsBindableVariable* [2];
   var_arc_d_t4b[0] = new ParcialBindableVariable(variables[fv +
0], 0);
   var_arc_d_t4b[1] = 0;
   inArcs_t4b[0] = new InArc<Sisprod>(
          this,
          places[fp + 3],
          &Sisprod::ArcExp_d_t4b,
          var_arc_d_t4b, 1);
   inArcs_t4b[1] = 0;
// transition t4b output arcs
   AbsOutArc** outArcs_t4b = new AbsOutArc*[3];
   outArcs_t4b[0] = new OutArc<Sisprod>(
          this,
          places[fp + 0],
          &Sisprod::ArcExp_t4b_a);
```

```
   outArcs_t4b[1] = new OutArc<Sisprod>(
          this,
          places[fp + 4],
          &Sisprod::ArcExp_t4b_e);
   outArcs_t4b[2] = 0;
   Action** actions_t4b = new Action* [1];
   actions_t4b[0] = 0;
// transition t4b
   transitions() = new Transition<Sisprod>(
          "t4b", inst,
          "Processar pecas 3 ou 4",
          inArcs_t4b, 1, outArcs_t4b,
          this,
          0, // code segment
          0, // guard
          pagEvents[fe + 4], // event
          actions_t4b, // actions
          -1); // priority
// ******************************
// transition t5 input arcs
   AbsInArc** inArcs_t5 = new AbsInArc*[3];
   const AbsBindableVariable** var_arc_e_t5 = new const
AbsBindableVariable* [2];
   var_arc_e_t5[0] = new ParcialBindableVariable(variables[fv +
1], 0);
   var_arc_e_t5[1] = 0;
   inArcs_t5[0] = new InArc<Sisprod>(
          this,
          places[fp + 4],
          &Sisprod::ArcExp_e_t5,
          var_arc_e_t5, 1);
   inArcs_t5[1] = new InArc<Sisprod>(
          this,
          places[fp + 0],
          &Sisprod::ArcExp_a_t5);
   inArcs_t5[2] = 0;
// transition t5 output arcs
   AbsOutArc** outArcs_t5 = new AbsOutArc*[2];
   outArcs_t5[0] = new OutArc<Sisprod>(
          this,
          places[fp + 5],
          &Sisprod::ArcExp_t5_f);
   outArcs_t5[1] = 0;
   Action** actions_t5 = new Action* [1];
   actions_t5[0] = 0;
// transition t5
   transitions() = new Transition<Sisprod>(
          "t5", inst,
          "Terminar processamento parcial",
          inArcs_t5, 2, outArcs_t5,
          this,
          0, // code segment
          &Sisprod::Guard_t5, // guard
          pagEvents[fe + 6], // event
          actions_t5, // actions
          -1); // priority
// ******************************
// transition t6 input arcs
   AbsInArc** inArcs_t6 = new AbsInArc*[2];
   const AbsBindableVariable** var_arc_f_t6 = new const
AbsBindableVariable* [2];
   var_arc_f_t6[0] = new ParcialBindableVariable(variables[fv +
0], 0);
   var_arc_f_t6[1] = 0;
   inArcs_t6[0] = new InArc<Sisprod>(
          this,
          places[fp + 5],
```

```
          &Sisprod::ArcExp_f_t6,
          var_arc_f_t6, 1);
   inArcs_t6[1] = 0;
// transition t6 output arcs
   AbsOutArc** outArcs_t6 = new AbsOutArc*[3];
   outArcs_t6[0] = new OutArc<Sisprod>(
          this,
          places[fp + 7],
          &Sisprod::ArcExp_t6_h);
   outArcs_t6[1] = new OutArc<Sisprod>(
          this,
          places[fp + 1],
          &Sisprod::ArcExp_t6_b);
   outArcs_t6[2] = 0;
   Action** actions_t6 = new Action* [1];
   actions_t6[0] = 0;
// transition t6
   transitions() = new Transition<Sisprod>(
          "t6", inst,
          "Por peca parcialmente processada no tapete seguinte",
          inArcs_t6, 1, outArcs_t6,
          this,
          0, // code segment
          &Sisprod::Guard_t6, // guard
          pagEvents[fe + 5], // event
          actions_t6, // actions
          -1); // priority
// ******************************
// transition t7 input arcs
   AbsInArc** inArcs_t7 = new AbsInArc*[2];
   const AbsBindableVariable** var_arc_e_t7 = new const
AbsBindableVariable* [2];
   var_arc_e_t7[0] = new ParcialBindableVariable(variables[fv +
1], 0);
   var_arc_e_t7[1] = 0;
   inArcs_t7[0] = new InArc<Sisprod>(
          this,
          places[fp + 4],
          &Sisprod::ArcExp_e_t7,
          var_arc_e_t7, 1);
   inArcs_t7[1] = 0;
// transition t7 output arcs
   AbsOutArc** outArcs_t7 = new AbsOutArc*[2];
   outArcs_t7[0] = new OutArc<Sisprod>(
          this,
          places[fp + 6],
          &Sisprod::ArcExp_t7_g);
   outArcs_t7[1] = 0;
   Action** actions_t7 = new Action* [1];
   actions_t7[0] = 0;
// transition t7
   transitions() = new Transition<Sisprod>(
          "t7", inst,
          "Por peca totalmente processada no tapete seguinte",
          inArcs_t7, 1, outArcs_t7,
          this,
          0, // code segment
          &Sisprod::Guard_t7, // guard
          pagEvents[fe + 6], // event
          actions_t7, // actions
          -1); // priority
// ******************************
// transition t8 input arcs
   AbsInArc** inArcs_t8 = new AbsInArc*[2];
   const AbsBindableVariable** var_arc_g_t8 = new const
AbsBindableVariable* [2];
```

```
   var_arc_g_t8[0] = new ParcialBindableVariable(variables[fv +
0], 0);
   var_arc_g_t8[1] = 0;
   inArcs_t8[0] = new InArc<Sisprod>(
          this,
          places[fp + 6],
          &Sisprod::ArcExp_g_t8,
          var_arc_g_t8, 1);
   inArcs_t8[1] = 0;
// transition t8 output arcs
   AbsOutArc** outArcs_t8 = new AbsOutArc*[2];
   outArcs_t8[0] = new OutArc<Sisprod>(
          this,
          places[fp + 7],
          &Sisprod::ArcExp_t8_h);
   outArcs_t8[1] = 0;
   Action** actions_t8 = new Action* [1];
   actions_t8[0] = 0;
// transition t8
   transitions() = new Transition<Sisprod>(
          "t8", inst,
          "Desocupar centro",
          inArcs_t8, 1, outArcs_t8,
          this,
          0, // code segment
          0, // guard
          pagEvents[fe + 5], // event
          actions_t8, // actions
          -1); // priority
// interfaces
//Initialization
   Init();
}


// places actions
bool Sisprod::Action0_b_moverTapete() {
#line 45 "sisprod.pag"
 return true;
}
bool Sisprod::Action0_c_pararTapete() {
#line 48 "sisprod.pag"
 return true;
}
bool Sisprod::Action0_d_alimentarComPeca() {
#line 51 "sisprod.pag"
 return true;
}
bool Sisprod::Action0_e_processarPeca() {
#line 54 "sisprod.pag"
 return true;
}
bool Sisprod::Action0_f_retirarPeca() {
#line 57 "sisprod.pag"
 return true;
}
bool Sisprod::Action0_g_retirarPeca() {
#line 60 "sisprod.pag"
 return true;
}

// guards
bool Sisprod::Guard_t1() const {
#line 65 "sisprod.pag"
 return i == 0;
}
bool Sisprod::Guard_t5() const {
```

```
#line 98 "sisprod.pag"

   return (ti.t < (nCentros-1));

}
bool Sisprod::Guard_t6() const {
#line 107 "sisprod.pag"

   return i < (nCentros-1);

}
bool Sisprod::Guard_t7() const {
#line 116 "sisprod.pag"

   return (ti.t == (nCentros-1));

}

// arc expressions
Expression Sisprod::ArcExp_a_t1() {
#line 67 "sisprod.pag"
 return new MultiSet<Centro>(i);
}
Expression Sisprod::ArcExp_t1_b() {
#line 69 "sisprod.pag"
 return new MultiSet<Centro>(i);
}
Expression Sisprod::ArcExp_b_t2() {
#line 73 "sisprod.pag"
 return new MultiSet<Centro>(i);
}
Expression Sisprod::ArcExp_t2_c() {
#line 75 "sisprod.pag"
 return new MultiSet<Centro>(i);
}
Expression Sisprod::ArcExp_c_t3() {
#line 79 "sisprod.pag"
 return new MultiSet<Centro>(i);
}
Expression Sisprod::ArcExp_h_t3() {
#line 80 "sisprod.pag"
 return new MultiSet<Centro>(i);
}
Expression Sisprod::ArcExp_t3_d() {
#line 82 "sisprod.pag"
 return new MultiSet<Centro>(i);
}
Expression Sisprod::ArcExp_d_t4a() {
#line 86 "sisprod.pag"
 return new MultiSet<Centro>(i);
}
Expression Sisprod::ArcExp_t4a_a() {
#line 88 "sisprod.pag"
 return new MultiSet<Centro>(i);
}
Expression Sisprod::ArcExp_t4a_e() {
#line 89 "sisprod.pag"
 return new MultiSet<TCentro>(TCentro(i, ttl1));
}
Expression Sisprod::ArcExp_d_t4b() {
#line 93 "sisprod.pag"
 return new MultiSet<Centro>(i);
}
Expression Sisprod::ArcExp_t4b_a() {
#line 95 "sisprod.pag"
 return new MultiSet<Centro>(i);
}
```

```
Expression Sisprod::ArcExp_t4b_e() {
#line 96 "sisprod.pag"
 return new MultiSet<TCentro>(TCentro(i, ttl2));
}
Expression Sisprod::ArcExp_e_t5() {
#line 102 "sisprod.pag"
 return new MultiSet<TCentro>(ti);
}
Expression Sisprod::ArcExp_a_t5() {
#line 103 "sisprod.pag"
 return new MultiSet<Centro>(Centro(ti.t + 1));
}
Expression Sisprod::ArcExp_t5_f() {
#line 105 "sisprod.pag"
 return new MultiSet<Centro>(Centro(ti.t));
}
Expression Sisprod::ArcExp_f_t6() {
#line 111 "sisprod.pag"
 return new MultiSet<Centro>(i);
}
Expression Sisprod::ArcExp_t6_h() {
#line 113 "sisprod.pag"
 return new MultiSet<Centro>(i);
}
Expression Sisprod::ArcExp_t6_b() {
#line 114 "sisprod.pag"
 return new MultiSet<Centro>(Centro(i + 1));
}
Expression Sisprod::ArcExp_e_t7() {
#line 120 "sisprod.pag"
 return new MultiSet<TCentro>(ti);
}
Expression Sisprod::ArcExp_t7_g() {
#line 122 "sisprod.pag"
 return new MultiSet<Centro>(Centro(ti.t));
}
Expression Sisprod::ArcExp_g_t8() {
#line 126 "sisprod.pag"
 return new MultiSet<Centro>(i);
}
Expression Sisprod::ArcExp_t8_h() {
#line 128 "sisprod.pag"
 return new MultiSet<Centro>(i);
}

// code segments

// code

#line 132 "sisprod.pag"

   int Sisprod::nCentros;
   int Sisprod::ttl1, Sisprod::ttl2;

   MultiSet<Sisprod::Centro> Sisprod::Inicializar() {
      MultiSet<Centro> mc;
      for(int i = 0; i < nCentros; i++)
         mc += MultiSet<Centro>(Centro(i));
      return mc;
   }
```

Ficheiro *sisprod.c* gerado pelo pré-processador a partir do código do Quadro 5.13.

Bibliografia

[Aho et al., 86] Aho, Alfred V., Sethi, Ravi e Ullman, Jeffrey D., *Compilers Principles, Techniques and Tools*, Addison-Wesley Publishing Company, 1986.

[Bastide et al., 95] Bastide, Rémi e Palanque, Philippe, "A Petri Net Based Environment for the Design of Event-Driven Interfaces", in *Proceedings of the 16th International Conference on Application and Theory of Petri Nets*, Giorgio de Michelis Michel Diaz (Eds.), Turin, Italy, LNCS 935, June 1995.

[Ben-Ari, 82] Ben-Ari, M., *Principles of Concurrent Programming*, Prentice-Hall, 1982.

[Berthomieu et al., 91] Berthomieu, Bernard e Diaz, Michel, "Modeling and Verification of Time Dependent Systems Using Time Petri Nets*", IEEE Transactions on Software Engineering*, vol. 17, no. 3, March 1991.

[Bloomquist, 90] Bloomquist, Lee G., "Object-Oriented Petri Nets: An Experience in Developing Object-Oriented Tools", *Proceedings of the 1990 Summer Computer Simulation Conference*, Calgary, Alberta, Canada, July 16-18, 1990.

[Booch, 94] Booch, Grady, *Object-Oriented Analysis and Design with Applications*, 2nd edition, The Benjamin/Cummings Publishing Company, Inc., 1994.

[Cargill, 93] Cargill, Tom, "Dangling references (again)", *C++ Report*, May 1993.

[Cargill, 94a] Cargill, Tom, "Correctness vs. efficiency", *C++ Report*, January 1994.

[Cargill, 94b] Cargill, Tom, "Pearson's String Hash", *C++ Report*, Vol. 6, No. 7, September 1994.

[Cline, 96] Cline, Marshall P. , *C++ FAQs Lite*, http://www.cerfnet.com/~mpcline/On-Line-C++-FAQ, 1996.

[Coplien, 92] Coplien, James O., *Advanced C++ Programming Styles and Idioms*, Addison-Wesley Publishing Company, 1992.

[Cormen et al., 90] Cormen, Thomas H., Leiserson, Charles E., Rivest, Ronald L., *Introduction to Algorithms*, second printing, MIT Press, 1990.

[David, 91] David, René, "Modeling of Dynamic Systems by Petri Nets", ECC 91 European Control Conference, Grenoble, France, July 2-5 1991.

[David et al., 92] David, René e Alla, Hassane, *Petri Nets and Grafcet Tools for modelling discrete event systems*, Prentice Hall, 1992.

[Ellis et al., 90] Ellis, Margaret A. e Stroustrup, *The Annotated C++ Reference Manual*, Addison-Wesley Publishing Company, 1990.

[Genrich, 86] Genrich, H. J., "Predicate/Transition Nets" in [Jensen, et al., 91:3-43].

[Ghezzi et al., 91] Ghezzi, C., Mandrioli, D., Morasca, S. e Pezzè, M., "A unified high-level Petri net formalism for time-critical systems", *IEEE Transactions on Software Enginnering*, 17(2), February 1991.

[Gomes, 91] Gomes, Luís Filipe dos Santos, *Especificação de sistemas digitais*, trabalho de síntese, submetido para a prestação de provas de aptidão pedagógica e capacidade científica, Departamento de Informática, Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa, Novembro 1991.

[Gomes et al, 92] Gomes, Luís e Steiger-Garção, A., "Especificação e realização de controladores utilizando redes de Petri coloridas e sincronizadas integrando lógica imprecisa", Workshop Iberoamericano de Sistemas Autónomos em Robótica e CIM, Lisboa, 2 a 4 de Novembro de 1992. Publicado em Revista Robótica e Automação", nº 10, Noc. 1992.

[Gomes et al., 93] Gomes, Luís, Steiger-Garção, A., Gama, Luís e Correia, Nuno, "Programação de Controladores Utilizando Redes de Petri", *Proceedings das 4as Jornadas Nacionais de Projecto Planeamento e Produção Assistidos por Computador*, Lisboa, Maio 1993 e INGENIUM, nº 72, Julho/Agosto 1993.

[Gomes et al., 94] Gomes, Luís, Steiger-Garção, A., Barros, João P. e Próspero Luís, P., "Simulação de Controladores Especificados através de Redes de Petri", *Proceedings do 1º Encontro Português de Controlo Automático - Controlo 94*, Instituto Superior Técnico, Lisboa, Setembro de 1994.

[Gomes et al., 95] Gomes, Luís, Steiger-Garção, "Programmable Controller Design Based on a Synchronized Colored Petri Net Model and Integrating Fuzzy Reasoning ",in *Proceedings of the 16th International Conference on Application and Theory of Petri Nets*, Giorgio de Michelis Michel Diaz (Eds.), Turin, Italy, LNCS 935, June 1995.

[Harel, 87] Harel, David, "Statecharts: A Visual Formalism for Complex Systems", Science of Computer Programming, vol. 8, 1987.

[Harel, 88] Harel, David, "On Visual Formalisms", Communications of the ACM, Vol. 31, No. 5, May 1988.

[Ho, 87] Ho, Yu-Chi, "Performance Evaluation and Perturbation Analysis of Discrete Event Dynamic Systems", *IEEE Transactions on Automatic Control*, vol. AC-32, No. 7, July 1987.

[Ho, 92a] Ho, Yu-Chi (Ed.), *Discrete Event Dynamic Systems: Analysing Complexity and Performance in the Modern World*, IEEE PRESS, 1992.

[Ho, 92b] Ho, Yu-Chi, "Introduction" *in Discrete Event Dynamic Systems: Analysing Complexity and Performance in the Modern World*, IEEE PRESS, 1992.

[Huber et al., 90] Huber, P., Jensen, K., e Shapiro, R.M., "Hierarchies in Coloured Petri Nets", in [Jensen, et al., 91:215-43].

[IEC, 92] International Electrotechnical Comission, Draft Int. Standard, Feb. 14, IEC 1131: Programmable Controllers, Part 3: Programming Languages, 1992.

[Inan et al., 88] Inan, Kemal, Varaiya, Pravin, "Finitely Recursive Process Models for Discrete Event Systems", *IEEE Transactions on Automatic Control*, Vol. 33, No. 7, July 1988.

[Itmi, s.d.] Itmi, Mhamed, "Designing Generalized Petri Nets", s.d..

[Jensen, 87] Jensen, "Coloured Petri Nets", *Advances in Petri Nets 1986*, (W. Brauer, W. Reisig, G. Rozemberg Eds), Lecture Notes Computer Science 254, Berlin-Heidelberg-New York: Springer-Verlag.

[Jensen, 90] Jensen, K., "Coloured Petri Nets: A High Level Language for System Design and Analysis", *in* [Jensen, et al., 91:342-417].

[Jensen, et al., 91] Jensen, K. e Rozemberg(Eds.) *High-level Petri Nets theory and application*, Spinger-Verlag, Berlin Heidelberg, 1991.

[Jensen, 92] Jensen, Kurt, *Coloured Petri Nets Basic Concepts, Analysis Methods and Practical Use*, Volume 1, Springer-Verlag, Berlin Heidelberg, 1992.

[Jensen, 94] Jensen, Kurt, "An Introduction to the Theoretical Aspects of Coloured Petri Nets", *A Decade of Concurrency*, Lecture Notes in Computer Science vol. 803, Springer-Verlag 1994, pp. 230-72.

[Jensen, 95] Jensen, Kurt, *Coloured Petri Nets Basic Concepts, Analysis Methods and Practical Use*, Volume 2, Springer-Verlag, Berlin Heidelberg, 1995.

[Keen et al., 93] Keen, C. D., Lakos, C. A., "A Methodology for the Construction of Simulation Models using Object-Oriented Petri Nets", paper presented at the *1993 European Simulation Multiconference on Adapting Object-Oriented Development Methodologies to the Design of Discrete Event Simulation Models*, 1993. Disponível em http://www.cs.utas.edu.au/Research/ opn.html.

[Kernigham et al., 88] Kernighan, Brian W. e Ritchie, Dennis M., *The C Programming Language*, second edition, AT&T Bell Laboratories, Murray Hill, New Jersey, Prentice Hall, Englewood Cliffs, New Jersey, E.U.A..

[Kumar et al., 94] Kumar, Devendra e Harous, Saad, "Distributed Simulation of Timed Petri Nets: Basic Problems and Their Resolution", *IEEE Trransactions on Systems, Man, and Cybernetics*, vol. 24, no.10, October 1994.

[Kruse, 84] Kruse, Robert L., *Data Structures and Program Design*, second edition, Prentice-Hall International , Inc., London, 1984.

[Lakos et al., 91] Lakos, C. A., Keen, C. D., "Simulation with Object-Oriented Petri Nets", paper presented at the Australian Software Engineering Conference, Sydney, July 1991. Disponível em http://www.cs.utas.edu.au/Research/opn.html.

[Lakos, 92a] Lakos, C. A. "LOOPN User Manual", March 1992. Dísponível em http://www.cs.utas.edu.au/Research/opn.html.

[Lakos, 92b] Lakos, C. A. "LOOPN System Manual", April 1992. Dísponível em http://www.cs.utas.edu.au/Research/opn.html.

[Lakos et al., 94] Lakos, C.A., Keen, C. D., "LOOPN++: A New Language for Object-Oriented Petri Nets", April 1994. Dísponível em http://www.cs.utas.edu.au/Research/opn.html.

[Lakos, 96] Lakos, Charles, "From Coloured Petri nets to Object Petri Niets", in *Proceedings of the 17th International Conference on Application and Theory of Petri Nets*, Jonhathan Billington, Wolfgang Reifig (Eds.), Osaka, Japan, LNCS 1091, June 1996.

[Levine et al., 92] Levine, John R., Manson, Tony e Brown, Doug, *lex & yacc*, second edition, O'Reilly & Associates, Inc, 1992.

[Lindqvist, 90] Lindqvist, M., "Parametrized Reachability Trees for Predicate/ Transition Nets" *in* [Jensen, et al., 91:351-72].

[Lippman, 91] Lippman, S. B., *A C++ Primer*, second edition, Addison-Wesley, 1991.

[Lucas, 93] Lucas, Paul Jay, *An Object-Oriented Language System for Implementing Concurrent, Hierarchical, Finite State Machines*, Submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science in the Graduate College of the University of Illinois at Urbana-Champaign, 1993.

[Manduchi et al., 94] Manduchi, G., Moro, M., "An Object Oriented Approach in Building an Environment for Simulation and Analysis Based on Timed Petri Nets with Multiple Execution Policies", IEEE, 1994.

[Marsan et al., 84] Marsan, A. M., Balbo, G., e Conte, G., "A class of generalized stochastic Petri nets for the performance evaluation of multiprocessor systems", ACM Trans. Comput. Syst., vol. 2, no. 2, pp. 93-122, May 1984.

[Marsan et al., 86] Marsan, A. M., Balbo, G., e Conte, G., *Performance Evaluation of Multiprocessor Systems using Petri Nets*. Cambridge, M.A.: MIT, 1986.

[Merlin et al., 76] Merlin, P. M. e Farber, D. J., "Recoverability of Communication Protocols", *IEEE Transactions on Communications*, vol. COMM-24.

[Meyer, 88] Meyer, Bertrand, *Object-oriented Software Construction*, Prentice-Hall International (UK) Ltd., 1988.

[Meyers, 92] Meyers, Scott, *Effective C++: 50 specific Ways to Improve Your Programs and Designs*, Addison-Wesley, 1992.

[Meyers, 96] Meyers, Scott, *More effective C++: 35 new Ways to Improve Your Programs and Designs*, Addison-Wesley, 1996.

[Milner et al., 90] Milner, R., Tofte, M., e Harper, R., *The Definition of Standard ML*, MIT Press, Cambridge, MA, 1990.

[Molloy et al., 82] Molloy, M. K., "Performance analysis using stochastic Petri nets", IEEE Trans. Comput., vol. 3, pp. 913-7, 1982.

[Morasca et al., 91] Morasca, Sandro, Pezzé, Mauro e Trubian, Marco, "Timed High-Level Nets", *The Journal of Real-Time Systems*, vol. 3, no. 2, pp. 165-89, May 1991.

[Murata, 89] Murata, Tadao, "Petri Nets: Properties, Analysis and Applications", *Proceedings of the IEEE*, Vol. 77, No. 4, April 1989.

[Murata, 95] Murata, Tomohiro, "Application of Petri Nets to Sequence Control Programming", *in* [Zhou, 95: 43-68]

[Murray, 93] Murray, Robert B., *C++ Strategies and Tactics*, Addison-Wesley Publishing Company, 1993.

[Novais, 94] Novais, José, *Programação de Autómatos Método Grafcet*, 2.ª edição, Fundação Calouste Gulbenkian, Lisboa, 1994.

[Peterson, 77] Peterson, James L., "Petri Nets", *Computing Surveys*, Vol. 9, No. 3, September 1977.

[Peterson, 81] Peterson, James L., *Petri Net Theory and the Modelling of Systems*, Prentice-Hall, 1981.

[Pezzé, 94] Pezzé, Mauro, "Cabernet: A Customizable Environment for the Specification and Analysis of Real-Time Systems", Dipartimento di Elettronica e Informazione, Politecnico di Milano, Piazza Leonardo da Vinci 32, 20133 Milano, Italy, ftp://ipeset4.elet.polimi.it/dist/Cabernet, June 2, 1994,.

[Ramchandani, 74] Ramchandani, C., "Analysis of asynchronous concurrent systems by timed Petri nets", Project MAC Technical Report MAC-TR-120, Massachusetts Institute of Technology, Cambridge MA, 1974.

[Raymond et al., s.d.] Raymond, Serge & Paludetto, Mario, Université Paul Sabatier/ LAAS-CNRS, Toulouse, France, "C++ et le Paralelism pour une Methode Orientees Objects", s.d..

[Reisig, 92] Reisig, W., *A Primer in Petri Net Design*, Springer-Verlag, Berlin Heidelberg, 1992.

[Sedgewick, 88] Sedgewick, Robert, *Algorithms*, second edition, Addison-Wesley Publishing Company, 1988.

[Sethi, 90] Sethi, Ravi, *Programming Languages Concepts and Constructs*, AT&T, Addison-Wesley Publishing Company, 1990.

[Schöf et al., 95] Schöf, Stefan, Sonnenschein, Michael e Wieting, Ralf, "Efficient Simulaion of THOR Nets", in *Proceedings of the 16th International Conference on Application and Theory of Petri Nets*, Giorgio de Michelis Michel Diaz (Eds.), Turin, Italy, LNCS 935, June 1995.

[Shukla et al., 91] Shukla, Ashish, Robbi, Anthony D., "A Petri Net Simulation Tool", Conference Proccedings 1991 IEEE International Conference on Systems, Man and Cybernetics, "Decision Aiding for Complex Systems, vol. 1, IEEE, New York, NY, USA, 1991.

[Silva, 85] Silva, Manuel, *Las Redes de Petri en la Automática y la Informática*, Madrid, Editorial AC, 1985.

[Stroustrup, 91] Stroustrup, Bjarne, *The C++ Programming Language*, 2nd edition, AT&T, Addison-Wesley Publishing Company, 1991.

[Stroustrup, 94] Stroustrup, Bjarne, *The Design and Evolution of C++*, AT&T, Addison-Wesley Publishing Company, 1994.

[Tichy et al., 94] Tichy, Walter F., Heilig, Jörg e Paulish, Frances Newbery, "A Generative and Generic approach to Persistence", *C++ Report*, Vol. 5, No. 10, January 1994.

[Venkatesh et al., 95] Venkatesh, Kurapati, Zhou, MengChu, e Caudill, Reggie J., "Discrete Event Control Design for Manufacturing Systems Via Ladder Logic Diagrams and Petri Nets: A Comparative Study", in [Zhou, 95], 1995.

[Wongtaladkown et al., 90] Wongtaladkown, Chanat, Chen, Yaobin, "A Real-Time Petri Nets Simulator for Automated Manufacturing Systems", *Proceedings of the 5th IEEE International Symposium on Intelligent Control 1990*, vol. 2, IEEE Comput. Soc. Press, Los Alamitos, CA, USA, pp. 999-1004, 1990.

[Wood et al., 90] Wood, Scott D. e Harsch II, Joseph C., "Object-Oriented Implementation of Parallel System Evaluation Tools: A Graphical Petri Net Simulation", Proceedings of the ISMM International Symposium Computer Applications in Design, Simulation and Analysis, MIMI' 90, Eds. Park, E. K., Acta Press, Anaheim, CA, USA, pp. 52-5, 1990.

[Zhou, 92] Zhou, MengChu, "Computer-Aided Modeling, Analysis, and Design of Discrete Event systems Using Petri Nets", *Proceedings of the International Symposium on Computer-Aided Control System Design*, Napa, California, March 17-19, 1992, pp. 255-61.

[Zhou, 95] Zhou, MengChu (Ed.), *Petri Nets in Flexible and Agile Automation*, Kluwer Academic Publishers, 1995.

[Zuberek, 91] Zuberek, W. M., "Timed Petri Nets Definitions, Properties, and Applications", *Microelectron. Reliab.*, Vol. 31, No. 4, pp. 627-44, 1991.

[Zurawsky et al., 94] Zurawsky, Richard e Zhou, MengChu, "Petri Nets and Industrial Applications: A Tutorial", *IEEE Transactions on Industrial Electronics*, Vol. 41, No. 6, December 1994.

Agradecimentos

Ao Professor Doutor Steiger Garção por todo o apoio e suporte disponibilizados ao longo dos últimos anos e que tornaram esta dissertação possível.

Ao Eng. Luís Gomes cujas sugestões constituiram um grande contributo para esta dissertação. A sua disponibilidade praticamente constante e os seus proveitosos conselhos foram uma enorme ajuda para o meu trabalho. Sem ele esta dissertação seria, certamente, menos interessante.

A MengChu Zhou, W. M. Zuberek, Mhamed Itmi e David Hill. Todos, gentilmente, me enviaram as publicações que lhes pedi, e ainda mais algumas.

A todos os que contribuíram para a *Petri Nets mailing List* durante a realização desta tese.

Ao João Pedro Neto que me obrigou a fazer as cadeiras muito mais depressa do que eu queria!

Ao Carlos Soares, rapaz de muitos ofícios, e companheiro constante (?!) desta aventura.

À Anikó Costa que me foi aturando ao longo de todo este tempo (e vice-versa).

Ao pessoal do CRI e UNINOVA e em especial ao Pedro Próspero Luís por todo apoio prestado.

À Rubina e ao João Carlos Silva pela conversa.

E à Guida que continua a ter muita paciência.

Proveniência das citações

A frase de Bertrand Russel que introduz "Simbologia e notações" é uma tradução livre da frase *Mathematics may be defined as the subject in which we never know what we are talking about, nor whether what we are saying is true*, do capítulo 4 do artigo "Mysticism and Logic", publicado originalmente em "International Monthly", vol.4, 1901. A famosa frase do Monty Python's Flying Circus é (na verdade): *And now for something completely different*. A frase de Tenesse Williams que introduz as redes de Petri com temporizações, é uma tradução de: *For time is the longest distance between two places*, retirada da obra *The Glass Menagerie* de 1994. *nimium ne crede œlori* significa *não confies demasiado na cor*, e *plus valet actum quam scriptum* significa *mais vale o que se fez do que o que se escreveu*. As observações de Feynman foram retiradas do livro *Uma tarde com o Sr. Feynman* da Editora Gradiva, pp. 23. Têm ambas origem numa conferência realizada em 1966 na 14.ª Convenção Anual da National Science Teacher Association e publicada em "Physics Teacher", Setembro de 1969, pp. 313-20. O texto da autoria de Fernando Pessoa é um excerto do poema "Liberdade". A frase do Hobbes foi retirada de uma tira publicada no jornal *Público* de 8 de Janeiro de 1995.